e300 Power Architecture™ Core Family Reference Manual

Supports
e300c1
e300c2
e300c3

e300coreRM
Rev. 3
12/2006

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
+1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org. The PowerPC name is a trademark of IBM Corp. and is used under license. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc., 2006. All rights reserved.

Document Number: e300coreRM
Rev. 3, 12/2006

Contents

Paragraph Number	Title	Page Number
	About This Book	
	Audience	xxiv
	Organization	xxiv
	Suggested Reading	xxv
	General Information	xxv
	Related Documentation	xxvi
	Conventions	xxvi
	Acronyms and Abbreviations	xxvii
	Chapter 1 Overview	
1.1	Overview	1-1
1.1.1	Features	1-5
1.1.2	Instruction Unit	1-8
1.1.2.1	Instruction Queue and Dispatch Unit	1-8
1.1.2.2	Branch Processing Unit (BPU)	1-9
1.1.3	Independent Execution Units	1-9
1.1.3.1	Integer Unit (IU)	1-9
1.1.3.2	Floating-Point Unit (FPU)	1-9
1.1.3.3	Load/Store Unit (LSU)	1-10
1.1.3.4	System Register Unit (SRU)	1-10
1.1.4	Completion Unit	1-10
1.1.5	Memory Subsystem Support	1-11
1.1.5.1	Memory Management Units (MMUs)	1-11
1.1.5.2	Cache Units	1-12
1.1.6	Bus Interface Unit (BIU)	1-12
1.1.7	System Support Functions	1-13
1.1.7.1	Power Management	1-13
1.1.7.2	Time Base/Decrementer	1-14
1.1.7.3	JTAG Test and Debug Interface	1-14
1.1.7.4	Clock Multiplier	1-14
1.1.7.5	Core Performance Monitor	1-14
1.2	PowerPC Architecture Implementation	1-15
1.3	Implementation-Specific Information	1-15
1.3.1	Register Model	1-16
1.3.1.1	UISA Registers	1-18
1.3.1.1.1	General-Purpose Registers (GPRs)	1-18

Paragraph Number	Title	Page Number
1.3.1.1.2	Floating-Point Registers (FPRs)	1-18
1.3.1.1.3	Condition Register (CR)	1-18
1.3.1.1.4	Floating-Point Status and Control Register (FPSCR)	1-18
1.3.1.1.5	User-Level SPRs	1-18
1.3.1.2	VEA Registers	1-19
1.3.1.3	OEA Registers	1-19
1.3.1.3.1	Machine State Register (MSR)	1-19
1.3.1.3.2	Segment Registers (SRs)	1-19
1.3.1.3.3	Supervisor-Level SPRs	1-19
1.3.2	Instruction Set and Addressing Modes	1-21
1.3.2.1	PowerPC Instruction Set and Addressing Modes	1-21
1.3.2.2	Implementation-Specific Instruction Set	1-22
1.3.3	Cache Implementation	1-23
1.3.3.1	PowerPC Cache Characteristics	1-23
1.3.3.2	Implementation-Specific Cache Organization	1-23
1.3.3.3	Instruction and Data Cache Way-Locking	1-25
1.3.4	Interrupt Model	1-25
1.3.4.1	PowerPC Interrupt Model	1-25
1.3.4.2	Implementation-Specific Interrupt Model	1-27
1.3.5	Memory Management	1-29
1.3.5.1	PowerPC Memory Management	1-29
1.3.5.2	Implementation-Specific Memory Management	1-30
1.3.6	Instruction Timing	1-30
1.3.7	Core Interface	1-31
1.3.7.1	Memory Accesses	1-32
1.3.7.2	Signals	1-33
1.3.8	Debug Features	1-33
1.3.8.1	Breakpoint Signaling	1-33
1.4	Differences Between Cores	1-34
1.5	Differences Between e300 Cores	1-36

Chapter 2
Register Model

2.1	PowerPC Register Set	2-1
2.2	Implementation-Specific Registers	2-10
2.2.1	Hardware Implementation Register 0 (HID0)	2-12
2.2.2	Hardware Implementation Register 1 (HID1)	2-15
2.2.3	Hardware Implementation Register 2 (HID2)	2-16
2.2.4	Data and Instruction TLB Miss Address Registers (DMISS and IMISS)	2-18

Paragraph Number	Title	Page Number
2.2.5	Data and Instruction TLB Compare Registers (DCMP and ICMP)	2-18
2.2.6	Primary and Secondary Hash Address Registers (HASH1 and HASH2)	2-19
2.2.7	Required Physical Address Register (RPA)	2-20
2.2.8	BAT Registers (BAT4–BAT7)	2-20
2.2.9	Critical Interrupt Save/Restore Register 0 (CSRR0)	2-21
2.2.10	Critical Interrupt Save/Restore Register 1 (CSRR1)	2-22
2.2.11	SPRG0–SPRG7	2-22
2.2.12	System Version Register (SVR)	2-23
2.2.13	System Memory Base Address (MBAR)	2-23
2.2.14	Instruction Address Breakpoint Registers (IABR and IABR2)	2-23
2.2.15	Instruction Address Breakpoint Control Register (IBCR)	2-24
2.2.16	Data Address Breakpoint Register (DABR and DABR2)	2-25
2.2.17	Data Address Breakpoint Control Register (DBCR)	2-27
2.2.18	Performance Monitor Registers	2-28

Chapter 3
Instruction Set Model

3.1	Operand Conventions	3-1
3.1.1	Data Organization in Memory and Memory Operands	3-1
3.1.2	Endian Modes and Byte Ordering	3-1
3.1.3	Alignment and Misaligned Accesses	3-2
3.1.4	Floating-Point Execution Model	3-3
3.1.5	Effect of Operand Placement on Performance	3-4
3.2	Instruction Set Summary	3-4
3.2.1	Classes of Instructions	3-5
3.2.1.1	Definition of Boundedly Undefined	3-5
3.2.1.2	Defined Instruction Class	3-6
3.2.1.3	Illegal Instruction Class	3-6
3.2.1.4	Reserved Instruction Class	3-7
3.2.2	Addressing Modes	3-7
3.2.2.1	Memory Addressing	3-7
3.2.2.2	Memory Operands	3-7
3.2.2.3	Effective Address Calculation	3-8
3.2.2.4	Synchronization	3-8
3.2.2.4.1	Context Synchronization	3-8
3.2.2.4.2	Execution Synchronization	3-9
3.2.2.4.3	Instruction-Related Interrupts	3-9
3.2.3	Instruction Set Overview	3-10

Paragraph Number	Title	Page Number
3.2.4	PowerPC UISA Instructions	3-10
3.2.4.1	Integer Instructions	3-10
3.2.4.1.1	Integer Arithmetic Instructions	3-10
3.2.4.1.2	Integer Compare Instructions	3-11
3.2.4.1.3	Integer Logical Instructions	3-12
3.2.4.1.4	Integer Rotate and Shift Instructions	3-12
3.2.4.2	Floating-Point Instructions	3-13
3.2.4.2.1	Floating-Point Arithmetic Instructions	3-14
3.2.4.2.2	Floating-Point Multiply-Add Instructions	3-14
3.2.4.2.3	Floating-Point Rounding and Conversion Instructions	3-15
3.2.4.2.4	Floating-Point Compare Instructions	3-15
3.2.4.2.5	Floating-Point Status and Control Register Instructions	3-15
3.2.4.2.6	Floating-Point Move Instructions	3-16
3.2.4.3	Load and Store Instructions	3-16
3.2.4.3.1	Self-Modifying Code	3-17
3.2.4.3.2	Integer Load and Store Address Generation	3-17
3.2.4.3.3	Register Indirect Integer Load Instructions	3-17
3.2.4.3.4	Integer Store Instructions	3-18
3.2.4.3.5	Integer Load and Store with Byte-Reverse Instructions	3-19
3.2.4.3.6	Integer Load and Store Multiple Instructions	3-19
3.2.4.3.7	Integer Load and Store String Instructions	3-20
3.2.4.3.8	Floating-Point Load and Store Address Generation	3-21
3.2.4.3.9	Floating-Point Load Instructions	3-21
3.2.4.3.10	Floating-Point Store Instructions	3-22
3.2.4.4	Branch and Flow Control Instructions	3-22
3.2.4.4.1	Branch Instruction Address Calculation	3-23
3.2.4.4.2	Branch Instructions	3-23
3.2.4.4.3	Condition Register Logical Instructions	3-23
3.2.4.5	Trap Instructions	3-24
3.2.4.6	Processor Control Instructions	3-24
3.2.4.6.1	Move To/From Condition Register Instructions	3-24
3.2.4.7	Memory Synchronization Instructions—UISA	3-25
3.2.5	PowerPC VEA Instructions	3-26
3.2.5.1	Processor Control Instructions	3-26
3.2.5.2	Memory Synchronization Instructions—VEA	3-27
3.2.5.3	Memory Control Instructions—VEA	3-27
3.2.6	PowerPC OEA Instructions	3-28
3.2.6.1	System Linkage Instructions	3-28
3.2.6.2	Processor Control Instructions—OEA	3-28
3.2.6.2.1	Move To/From Machine State Register Instructions	3-29
3.2.6.2.2	Move To/From Special-Purpose Register Instructions	3-29

3.2.6.2.3 Move To/From Performance Monitor Register Instructions 3-32
3.2.6.3 Memory Control Instructions—OEA 3-32
3.2.6.3.1 Supervisor-Level Cache Management Instruction 3-32
3.2.6.3.2 Segment Register Manipulation Instructions 3-32
3.2.6.3.3 Translation Lookaside Buffer Management Instructions 3-33
3.2.7 Recommended Simplified Mnemonics 3-33
3.2.8 Implementation-Specific Instructions 3-34

Chapter 4
Instruction and Data Cache Operation

4.1 Introduction 4-1
4.1.1 Instruction and Data Cache Features 4-1
4.1.2 Overview 4-2
4.2 Data Cache Organization 4-3
4.3 Instruction Cache Organization 4-4
4.4 Memory and Cache Coherency 4-5
4.4.1 Memory/Cache Access Attributes (WIMG Bits) 4-6
4.4.1.1 Write-Through Attribute (W) 4-6
4.4.1.2 Caching-Inhibited Attribute (I) 4-7
4.4.1.3 Memory Coherency Attribute (M) 4-7
4.4.1.4 Guarded Attribute (G) 4-7
4.4.1.5 W, I, and M Bit Combinations 4-8
4.4.2 Coherency Support 4-9
4.4.2.1 MEI Coherency Protocol 4-9
4.4.2.1.1 MEI State Transitions 4-9
4.4.2.2 MESI Coherency Protocol 4-10
4.4.2.2.1 MESI State Transitions 4-11
4.4.2.3 Load and Store Coherency Summary 4-12
4.4.2.4 Coherency in Single-Processor Systems 4-12
4.4.3 Core-Initiated Load/Store Operations 4-13
4.4.3.1 Performed Loads and Stores 4-13
4.4.3.2 Sequential Consistency of Memory Accesses 4-13
4.4.3.3 Enforcing Load/Store Ordering 4-14
4.4.3.4 Atomic Memory References 4-14
4.5 Cache Control 4-14
4.5.1 Cache Control Parameters in HID0 and HID2 4-14
4.5.1.1 Cache Parity Error Reporting—HID0[ECPE] 4-14
4.5.1.2 Data Cache Enable—HID0[DCE] 4-14
4.5.1.3 Data Cache Lock—HID0[DLOCK] 4-15
4.5.1.4 Data Cache Way-lock—HID2[DWLCK] 4-15

Paragraph Number	Title	Page Number
4.5.1.5	Data Cache Flash Invalidate—HID0[DCFI]	4-15
4.5.1.6	Instruction Cache Enable—HID0[ICE]	4-16
4.5.1.7	Instruction Cache Lock—HID0[ILOCK]	4-16
4.5.1.8	Instruction Cache Way-Lock—HID2[IWLCK]	4-16
4.5.1.9	Instruction Cache Flash Invalidate—HID0[ICFI]	4-16
4.5.1.10	Instruction Cache Way Protect—HID2[ICWP]	4-17
4.5.1.11	Cache Operation Broadcasting—HID0[ABE]	4-17
4.5.2	Cache Control Instructions	4-17
4.5.2.1	Data Cache Block Touch (**dcbt**) Instruction	4-18
4.5.2.2	Data Cache Block Touch For Store (**dcbtst**) Instruction	4-18
4.5.2.3	Data Cache Block Clear To Zero (**dcbz**) Instruction	4-18
4.5.2.4	Data Cache Block Store (**dcbst**) Instruction	4-19
4.5.2.5	Data Cache Block Flush (**dcbf**) Instruction	4-19
4.5.2.6	Data Cache Block Invalidate (**dcbi**) Instruction	4-19
4.5.2.7	Instruction Cache Block Touch (**icbt**) Instruction	4-20
4.5.2.8	Instruction Cache Block Invalidate (**icbi**) Instruction	4-20
4.6	Cache Operations	4-21
4.6.1	Data Cache Fill Operations	4-21
4.6.2	Instruction Cache Fill Operations	4-21
4.6.3	Instruction Fetch Cancel Extension	4-21
4.6.4	Data Cache Cast-Out Operations	4-21
4.6.5	Cache Block Push Operation	4-22
4.6.6	Data Cache Queue Sharing Extension	4-22
4.6.7	Cache Block Replacement Selection	4-22
4.7	L1 Cache Parity	4-25
4.8	Bus Interface	4-26
4.9	Caches and CSB Transactions	4-27
4.9.1	Single-Beat Transactions	4-27
4.9.2	Burst Transactions	4-27
4.9.3	Instruction Fetch Burst Enable for Caching-Inhibited Space	4-28
4.9.4	CSB Operations Caused by Cache Control Instructions	4-29
4.9.5	Snooping	4-29
4.10	Applications Information—Cache Locking	4-32
4.10.1	Cache Locking Terminology	4-32
4.10.2	Cache Locking Register Summary	4-33
4.10.3	Performing Cache Locking	4-34
4.10.3.1	Data Cache Locking—Procedures	4-35
4.10.3.1.1	Enabling the Data Cache	4-35
4.10.3.1.2	Address Translation for Data Cache Locking	4-35
4.10.3.1.3	Disabling Interrupts for Data Cache Locking	4-36
4.10.3.1.4	Invalidating the Data Cache	4-36

4.10.3.1.5 Loading the Data Cache 4-37
4.10.3.1.6 Locking the Entire Data Cache 4-38
4.10.3.1.7 Way-Locking the Data Cache 4-38
4.10.3.1.8 Invalidating the Data Cache (Even if Locked) 4-39
4.10.3.2 Instruction Cache Locking—Procedures 4-39
4.10.3.2.1 Enabling the Instruction Cache 4-40
4.10.3.2.2 Address Translation for Instruction Cache Locking 4-40
4.10.3.2.3 Disabling Interrupts for Instruction Cache Locking 4-41
4.10.3.2.4 Preloading Instructions into the Instruction Cache 4-41
4.10.3.2.5 Locking the Entire Instruction Cache 4-43
4.10.3.2.6 Way-Locking the Instruction Cache 4-43
4.10.3.2.7 Invalidating the Instruction Cache (Even if Locked) 4-44
4.10.3.2.8 Instruction Cache Way Protection 4-45

Chapter 5
Interrupts and Exceptions

5.1 Interrupt Classes 5-2
5.1.1 Interrupt Priorities 5-6
5.1.2 Summary of Front-End Interrupt Handling 5-7
5.2 Interrupt Processing 5-8
5.2.1 Interrupt Processing Registers 5-8
5.2.1.1 SRR0 and SRR1 Bit Settings 5-8
5.2.1.2 CSRR0 and CSRR1 Bit Settings 5-10
5.2.1.3 SPRG0–SPRG7 5-11
5.2.1.4 MSR Bit Settings 5-12
5.2.2 Enabling and Disabling Exceptions and Interrupts 5-14
5.2.3 Steps for Interrupt Processing 5-15
5.2.4 Setting MSR[RI] 5-15
5.2.5 Returning from an Interrupt with **rfi** 5-15
5.2.6 Returning from an Interrupt with **rfci** 5-16
5.3 Process Switching 5-16
5.4 Interrupt Latencies 5-16
5.5 Interrupt Definitions 5-17
5.5.1 Reset Interrupts (0x00100) 5-18
5.5.1.1 Hard Reset and Power-On Reset 5-18
5.5.1.2 Soft Reset 5-19
5.5.1.3 Byte Ordering Considerations 5-20
5.5.2 Machine Check Interrupt (0x00200) 5-21
5.5.2.1 Machine Check Interrupt Enabled (MSR[ME] = 1) 5-22
5.5.2.2 Checkstop State (MSR[ME] = 0) 5-22

Paragraph Number	Title	Page Number
5.5.3	DSI Interrupt (0x00300)	5-23
5.5.4	ISI Interrupt (0x00400)	5-24
5.5.5	External Interrupt (0x00500)	5-25
5.5.6	Alignment Interrupt (0x00600)	5-25
5.5.6.1	Integer Alignment Exceptions	5-27
5.5.6.2	Load/Store Multiple Alignment Exceptions	5-28
5.5.7	Program Interrupt (0x00700)	5-28
5.5.7.1	IEEE Floating-Point Exception Program Interrupts	5-29
5.5.7.2	Illegal, Reserved, and Unimplemented Instructions Program Interrupts	5-29
5.5.8	Floating-Point Unavailable Interrupt (0x00800)	5-29
5.5.9	Decrementer Interrupt (0x00900)	5-30
5.5.10	Critical Interrupt (0x00A00)	5-30
5.5.11	System Call Interrupt (0x00C00)	5-31
5.5.12	Trace Interrupt (0x00D00)	5-31
5.5.12.1	Single-Step Instruction Trace Mode	5-32
5.5.12.2	Branch Trace Mode	5-32
5.5.13	Performance Monitor Interrupt (0x00F00)	5-33
5.5.14	Instruction TLB Miss Interrupt (0x01000)	5-33
5.5.15	Data TLB Miss on Load Interrupt (0x01100)	5-33
5.5.16	Data TLB Miss on Store Interrupt (0x01200)	5-34
5.5.17	Instruction Address Breakpoint Interrupt (0x01300)	5-34
5.5.18	System Management Interrupt (0x01400)	5-36

Chapter 6
Memory Management

6.1	MMU Features	6-2
6.1.1	Memory Addressing	6-3
6.1.2	MMU Organization	6-3
6.1.3	Address Translation Mechanisms	6-8
6.1.4	Memory Protection Facilities	6-9
6.1.5	Page History Information	6-10
6.1.6	General Flow of MMU Address Translation	6-11
6.1.6.1	Real Addressing Mode and Block Address Translation Selection	6-11
6.1.6.2	Page Address Translation Selection	6-12
6.1.7	MMU Interrupts Summary	6-14
6.1.8	MMU Instructions and Register Summary	6-16
6.2	Real Addressing Mode	6-19
6.3	Block Address Translation	6-19
6.4	Memory Segment Model	6-19
6.4.1	Page History Recording	6-20

6.4.1.1 Reference Bit ... 6-21
6.4.1.2 Change Bit ... 6-21
6.4.1.3 Scenarios for Reference and Change Bit Recording ... 6-22
6.4.2 Page Memory Protection ... 6-23
6.4.3 TLB Description ... 6-23
6.4.3.1 TLB Organization ... 6-23
6.4.3.2 TLB Entry Invalidation ... 6-25
6.4.4 Page Address Translation Summary ... 6-25
6.5 Page Table Search Operation ... 6-25
6.5.1 Page Table Search Operation—Conceptual Flow ... 6-25
6.5.2 Implementation-Specific Table Search Operation ... 6-29
6.5.2.1 Resources for Table Search Operations ... 6-29
6.5.2.1.1 Data and Instruction TLB Miss Address Registers (DMISS and IMISS) ... 6-31
6.5.2.1.2 Data and Instruction TLB Compare Registers (DCMP and ICMP) ... 6-32
6.5.2.1.3 Primary and Secondary Hash Address Registers (HASH1 and HASH2) ... 6-32
6.5.2.1.4 Required Physical Address Register (RPA) ... 6-33
6.5.2.2 Software Table Search Operation ... 6-34
6.5.2.2.1 Flow for Example Interrupt Handlers ... 6-34
6.5.2.2.2 Code for Example Interrupt Handlers ... 6-38
6.5.3 Page Table Updates ... 6-44
6.5.4 Segment Register Updates ... 6-44

Chapter 7
Instruction Timing

7.1 Terminology and Conventions ... 7-1
7.2 Instruction Timing Overview ... 7-3
7.3 Timing Considerations ... 7-9
7.3.1 General Instruction Flow ... 7-9
7.3.2 Instruction Fetch Timing ... 7-10
7.3.2.1 Cache Arbitration ... 7-10
7.3.2.2 Cache Hit ... 7-11
7.3.2.3 Cache Miss ... 7-14
7.3.3 Instruction Dispatch and Completion Considerations ... 7-16
7.3.3.1 Rename Register Operation ... 7-16
7.3.3.2 Instruction Serialization ... 7-17
7.3.3.3 Execution Unit Considerations ... 7-17
7.4 Execution Unit Timings ... 7-17
7.4.1 Branch Processing Unit Execution Timing ... 7-18
7.4.1.1 Branch Folding ... 7-18
7.4.1.2 Static Branch Prediction ... 7-19

Paragraph Number	Title	Page Number
7.4.1.2.1	Predicted Branch Timing Examples	7-20
7.4.2	Integer Unit Execution Timing	7-21
7.4.3	Floating-Point Unit Execution Timing	7-23
7.4.4	Load/Store Unit Execution Timing	7-24
7.4.5	System Register Unit Execution Timing	7-24
7.5	Memory Performance Considerations	7-24
7.5.1	Copy-Back Mode	7-25
7.5.2	Write-Through Mode	7-25
7.5.3	Cache-Inhibited Accesses	7-25
7.6	Instruction Scheduling Guidelines	7-26
7.6.1	Branch, Dispatch, and Completion Unit Resource Requirements	7-26
7.6.1.1	Branch Resolution Resource Requirements	7-26
7.6.1.2	Dispatch Unit Resource Requirements	7-27
7.6.1.3	Completion Unit Resource Requirements	7-27
7.7	Instruction Latency Summary	7-28

Chapter 8
Core Interface Operation

8.1	Signal Groupings	8-1
8.1.1	Functional Groupings	8-2
8.1.2	Signal Summary	8-2
8.1.2.1	PLL Configuration (pll_cfg[0:6])—Input	8-4
8.2	Overview of Core Interface Accesses	8-7
8.3	Interrupt, Checkstop, and Reset Signals	8-8
8.3.1	External Interrupts	8-8
8.3.2	Checkstops	8-9
8.3.3	Reset Inputs	8-9
8.3.4	Core Quiesce Control Signals	8-9
8.4	IEEE 1149.1-Compliant Interface	8-9
8.4.1	IEEE 1149.1 Interface Description	8-9

Chapter 9
Power Management

9.1	Overview	9-1
9.2	Dynamic Power Management	9-1
9.3	Programmable Power Modes	9-1
9.3.1	Power Management Modes	9-3
9.3.1.1	Full-Power Mode with DPM Disabled	9-3
9.3.1.2	Full-Power Mode with DPM Enabled	9-3

Paragraph Number	Title	Page Number
9.3.1.3	Doze Mode	9-3
9.3.1.4	Nap Mode	9-4
9.3.1.5	Sleep Mode	9-4
9.3.2	Power Management Software Considerations	9-5
9.4	Example Code Sequence for Entering Processor Sleep Mode	9-6

Chapter 10
Debug Features

10.1	Breakpoint Resources	10-1
10.1.1	Instruction Address Breakpoint Registers (IABR, IABR2)	10-1
10.1.2	Instructional Address Control Register (IBCR)	10-2
10.1.3	Data Address Breakpoint Registers (DABR, DABR2)	10-2
10.1.4	Data Address Control Register (DBCR)	10-2
10.1.5	Other Debug Resources	10-2
10.1.6	Interrupt Vectors for Debugging	10-3
10.2	Using Breakpoint Facilities	10-3
10.2.1	Single-Stepping	10-3
10.2.2	Branch Tracing	10-4
10.2.3	Breakpoint Address Matching Options	10-4
10.3	Synchronization Requirements and Other Precautions	10-6

Chapter 11
Performance Monitor

11.1	Overview	11-1
11.2	Performance Monitor Registers	11-2
11.2.1	Global Control Register 0 (PMGC0)	11-3
11.2.2	User Global Control Register 0 (UPMGC0)	11-4
11.2.3	Local Control A Registers (PMLCa0–PMLCa3)	11-4
11.2.4	User Local Control A Registers (UPMLCa0–UPMLCa3)	11-5
11.2.5	Performance Monitor Counter Registers (PMC0–PMC3)	11-5
11.2.6	User Performance Monitor Counter Registers (UPMC0–UPMC3)	11-7
11.2.7	Performance Monitor Instructions	11-7
11.3	Performance Monitor Interrupt	11-9
11.4	Event Counting	11-10
11.4.1	Processor Context Configurability	11-10
11.5	Performance Monitor Application Examples	11-11
11.5.1	Chaining Counters	11-11
11.5.2	Event Selection	11-11

Paragraph Number	Title	Page Number

Appendix A
Instruction Set Listings

A.1 Instructions Sorted by Mnemonic A-1
A.2 Instructions Sorted by Opcode A-8
A.3 Instructions Grouped by Functional Categories A-15
A.4 Instructions Sorted by Form A-24
A.5 Instruction Set Legend A-35

Appendix B
Instructions Not Implemented

Appendix C
Revision History

C.1 Changes from Revision 2 to Revision 3 ..... C-1
C.2 Changes from Revision 1 to Revision 2 ..... C-1
C.3 Changes From Revision 0 to Revision 1 ..... C-4

Glossary

Index

Figures

Figure Number	Title	Page Number
1-1	e300c1 Core Block Diagram	1-2
1-2	e300c2 Core Block Diagram	1-3
1-3	e300c3 Core Block Diagram	1-4
1-4	e300 Programming Model—Registers	1-18
1-5	e300c1 Data Cache Organization	1-24
1-6	e300c2 and e300c3 Data Cache Organization	1-24
1-7	Core Interface	1-32
2-1	e300 Programming Model—Registers	2-3
2-2	Floating-Point Status and Control Register (FPSCR)	2-3
2-3	e300 Processor Version Register	2-6
2-4	Machine State Register	2-7
2-5	HID0 Register	2-12
2-6	HID1 Register	2-15
2-7	HID2 Register	2-16
2-8	DMISS and IMISS Registers	2-18
2-9	DCMP and ICMP Registers	2-19
2-10	HASH1 and HASH2 Registers	2-19
2-11	Required Physical Address Register (RPA)	2-20
2-12	Upper BAT Register	2-21
2-13	Lower BAT Register	2-21
2-14	Critical Interrupt Save/Restore Register 0 (CSRR0)	2-22
2-15	Critical Interrupt Save/Restore Register 1 (CSRR1)	2-22
2-16	Critical Interrupt Save/Restore Register 0 (CSRR0)	2-22
2-17	SPRG*n* Register	2-22
2-18	SVR Register	2-23
2-19	IABR and IABR2 Registers	2-24
2-20	IBCR Register	2-24
2-21	DABR and DABR2 Registers	2-25
2-22	DBCR Register	2-27
4-1	e300c1 Data Cache Organization	4-3
4-2	e300c2 and e300c3 Data Cache Organization	4-3
4-3	e300c1 Instruction Cache Organization	4-4
4-4	e300c2 and e300c3 Instruction Cache Organization	4-5
4-5	MEI Cache Coherency Protocol—State Diagram (WIM = 001)	4-10
4-6	MESI Cache Coherency Protocol—State Diagram (WIM = 001)	4-11
4-7	PLRU Replacement Algorithm	4-24
4-8	Bus Interface Address Buffers	4-27
4-9	Double-Word Address Ordering—Critical-Double-Word-First	4-28
5-1	Machine Status Save/Restore Register 0 (SRR0)	5-8

Figures

Figure Number	Title	Page Number
5-2	Machine Status Save/Restore Register 1 (SRR1)	5-8
5-3	Critical Interrupt Save/Restore Register 0 (CSRR0)	5-10
5-4	Critical Interrupt Save/Restore Register 1 (CSRR1)	5-11
5-5	SPRG*n* Register	5-11
5-6	Machine State Register (MSR)	5-12
6-1	MMU Conceptual Block Diagram—32-Bit Implementations	6-5
6-2	e300 Core IMMU Block Diagram	6-6
6-3	e300 Core DMMU Block Diagram	6-7
6-4	Address Translation Types	6-9
6-5	General Flow of Address Translation (Real Addressing Mode and Block)	6-11
6-6	General Flow of Page and Direct-Store Interface Address Translation	6-13
6-7	Segment Register and TLB Organization	6-24
6-8	Page Address Translation Flow for 32-Bit Implementations—TLB Hit	6-26
6-9	Primary Page Table Search—Conceptual Flow	6-28
6-10	Secondary Page Table Search Flow—Conceptual Flow	6-29
6-11	DMISS and IMISS Registers	6-32
6-12	DCMP and ICMP Registers	6-32
6-13	HASH1 and HASH2 Registers	6-33
6-14	Required Physical Address Register (RPA)	6-33
6-15	Flow for Example Software Table Search Operation	6-35
6-16	Check and Set R and C Bit Flow	6-36
6-17	Page Fault Setup Flow	6-37
6-18	Setup for Protection Violation Exceptions	6-38
7-1	Pipelined Execution Unit	7-3
7-2	Instruction Flow Diagram for the e300c1	7-4
7-3	Instruction Flow Diagram for the e300c2	7-5
7-4	Instruction Flow Diagram for the e300c3	7-7
7-5	e300 Core Processor Pipeline Stages	7-8
7-6	Instruction Timing—Cache Hit	7-12
7-7	Instruction Timing—Cache Miss	7-15
7-8	Branch Instruction Timing	7-21
7-9	Instruction Timing—Integer Execution in the e300c1Core	7-22
7-10	Instruction Timing—Integer Execution in the e300c2 and e300c3	7-23
8-1	Core Interface Signals	8-2
11-1	Performance Monitor Global Control Register 0 (PMGC0)/ User Performance Monitor Global Control Register 0 (UPMGC0)	11-3
11-2	Local Control A Registers (PMLCa0–PMLCa3)/ User Local Control A Registers (UPMLCa0–UPMLCa3)	11-4
11-3	Performance Monitor Counter Registers (PMC0–PMC3)/ User Performance Monitor Counter Registers (UPMC0–UPMC3)	11-6

Tables

Table Number	Title	Page Number
1-1	Interrupt Classifications	1-27
1-2	Exceptions and Interrupts	1-27
1-3	Differences Between e300 and G2_LE Cores	1-34
1-4	Differences Between e300 Cores	1-36
2-1	FPSCR Bit Settings	2-4
2-2	Architectural PVR Field Descriptions	2-6
2-3	Assigned PVR Values	2-7
2-4	MSR Bit Settings	2-7
2-5	e300 HID0 Field Descriptions	2-12
2-6	HID0[SBCLK] and HID0[ECLK] *clk_out* Configuration	2-15
2-7	HID1 Bit Settings	2-15
2-8	e300 HID2 Field Descriptions	2-16
2-9	DCMP and ICMP Bit Settings	2-19
2-10	HASH1 and HASH2 Bit Settings	2-19
2-11	RPA Bit Settings	2-20
2-12	System Version Register (SVR) Bit Settings	2-23
2-13	Instruction Address Breakpoint Register (IABR and IABR2) Bit Settings	2-24
2-14	Instruction Address Breakpoint Control Registers (IBCR)	2-24
2-15	Data Address Breakpoint Registers (DABR and DABR2) Bit Settings	2-26
2-16	Data Address Breakpoint Control Registers (DBCR)	2-27
3-1	Endian Mode Indication	3-2
3-2	Memory Operands	3-2
3-3	Integer Arithmetic Instructions	3-10
3-4	Integer Compare Instructions	3-11
3-5	Integer Logical Instructions	3-12
3-6	Integer Rotate Instructions	3-13
3-7	Integer Shift Instructions	3-13
3-8	Floating-Point Arithmetic Instructions	3-14
3-9	Floating-Point Multiply-Add Instructions	3-14
3-10	Floating-Point Rounding and Conversion Instructions	3-15
3-11	Floating-Point Compare Instructions	3-15
3-13	Floating-Point Move Instructions	3-16
3-14	Integer Load Instructions	3-17
3-15	Integer Store Instructions	3-18
3-16	Integer Load and Store with Byte-Reverse Instructions	3-19
3-17	Integer Load and Store Multiple Instructions	3-20
3-18	Integer Load and Store String Instructions	3-20
3-19	Floating-Point Load Instructions	3-21
3-20	Floating-Point Store Instructions	3-22
3-21	Branch Instructions	3-23
3-22	Condition Register Logical Instructions	3-24

Tables

Table Number	Title	Page Number
3-23	Trap Instructions	3-24
3-24	Move To/From Condition Register Instructions	3-24
3-25	Memory Synchronization Instructions—UISA	3-26
3-26	Move From Time Base Instruction	3-26
3-27	Memory Synchronization Instructions—VEA	3-27
3-28	User-Level Cache Instructions	3-27
3-29	System Linkage Instructions	3-28
3-30	Move To/From Machine State Register Instructions	3-29
3-31	Move To/From Special-Purpose Register Instructions	3-29
3-32	Implementation-Specific SPR Encodings (**mfspr**)	3-29
3-33	Performance Monitor APU Instructions	3-32
3-34	Segment Register Manipulation Instructions	3-32
3-35	Translation Lookaside Buffer Management Instructions	3-33
4-1	Combinations of W, I, and M Bits	4-8
4-2	MEI/MESI State Definitions	4-9
4-3	Memory Coherency Actions on Load Operations	4-12
4-4	Memory Coherency Actions on Store Operations	4-12
4-5	e300c1 PLRU Replacement Way Selection	4-23
4-6	e300c2 PLRU Replacement Way Selection	4-23
4-7	PLRU Bit Update Rules	4-25
4-8	e300 Bus Operations Caused by Cache Control Instructions	4-29
4-9	Snoop Response to CSB Transactions	4-30
4-10	Cache Organization	4-33
4-11	HID0 Bits Used to Perform Cache Locking	4-33
4-12	HID2 Bits Used to Perform Cache Way-locking	4-34
4-13	MSR Bits Used to Perform Cache Locking	4-34
4-14	Example BAT Settings for Cache Locking	4-35
4-15	MSR Bits for Disabling Interrupts	4-36
4-16	e300c1 Core DWLCK[0–2] Encodings	4-38
4-17	e300c2 and e300c3 Core DWLCK[0–2] Encodings	4-39
4-18	Example BAT Settings for Cache Locking	4-40
4-19	MSR Bits for Disabling Interrupts	4-41
4-20	e300c1 Core IWLCK[0–2] Encodings	4-43
4-21	e300c2 Core IWLCK[0–2] Encodings	4-44
5-1	Interrupt Classifications	5-3
5-2	Interrupts and Exception Conditions	5-3
5-3	Interrupt Priorities	5-6
5-4	SRR1 Bit Settings for Machine Check Interrupts	5-9
5-5	SRR1 Bit Settings for Program Interrupts	5-9
5-6	SRR1 Bit Settings for Software Table Search Operations	5-10
5-7	Conventional Uses of SPRG0–SPRG7	5-11

Tables

Table Number	Title	Page Number
5-8	MSR Bit Settings	5-12
5-9	IEEE Floating-Point Exception Mode Bits	5-14
5-10	MSR Setting Due to Interrupt	5-17
5-11	Hard Reset MSR Value and Interrupt Vector	5-18
5-12	Settings Caused by Hard Reset	5-19
5-13	Soft Reset Interrupt—Register Settings	5-20
5-14	Machine Check Interrupt—Register Settings	5-22
5-15	DSI Interrupt—Register Settings	5-23
5-16	External Interrupt—Register Settings	5-25
5-17	Alignment Interrupt—Register Settings	5-26
5-18	Access Types	5-27
5-19	Critical Interrupt—Register Settings	5-31
5-20	Trace Interrupt—Register Settings	5-32
5-21	Instruction and Data TLB Miss Interrupts—Register Settings	5-34
5-22	Instruction Address Breakpoint Interrupt—Register Settings	5-35
5-23	Breakpoint Action for Multiple Modes Enabled for the Same Address	5-35
5-24	System Management Interrupt—Register Settings	5-36
6-1	MMU Features Summary	6-2
6-2	Access Protection Options for Pages	6-9
6-3	Translation Exception Conditions	6-14
6-4	Other MMU Exception Conditions	6-15
6-5	Instruction Summary—MMU Control	6-17
6-6	MMU Registers	6-17
6-7	Table Search Operations to Update History Bits—TLB Hit Case	6-20
6-8	Model for Guaranteed R and C Bit Settings	6-22
6-9	Implementation-Specific Resources for Table Search Operations	6-30
6-10	Implementation-Specific SRR1 Bits	6-31
6-11	DCMP and ICMP Bit Settings	6-32
6-12	HASH1 and HASH2 Bit Settings	6-33
6-13	RPA Bit Settings	6-33
7-1	Branch Instructions	7-28
7-2	System Register Instructions	7-28
7-3	Condition Register Logical Instructions	7-29
7-4	Integer Instructions	7-29
7-5	Floating-Point Instructions	7-31
7-6	Load and Store Instructions	7-32
8-1	Summary of Selected Internal Signals	8-2
8-2	Core PLL Configuration	8-4
9-1	e300 Core Programmable Power Modes	9-2
10-1	Other Debug and Support Register Bits	10-3
10-2	Debug Interrupts and Conditions	10-3

Tables

Table Number	Title	Page Number
10-3	Single-Address Matching Bit Settings	10-5
10-4	Two-Address OR Matching	10-5
10-5	Address Matching for Inside Address Range	10-6
10-6	Address Matching for Outside Address Range	10-6
11-1	Performance Monitor Registers–Supervisor Level	11-2
11-2	Performance Monitor Registers–User Level (Read-Only)	11-2
11-3	PMGC0 Field Descriptions	11-3
11-4	PMLCa0–PMLCa3 Field Descriptions	11-5
11-5	PMC0–PMC3 Field Descriptions	11-6
11-6	Performance Monitor APU Instructions	11-7
11-7	Processor States and PMLCa0–PMLCa3 Bit Settings	11-10
11-8	Event Types	11-11
11-9	Performance Monitor Event Selection	11-12
A-1	Complete Instruction List Sorted by Mnemonic	A-1
A-2	Complete Instruction List Sorted by Opcode	A-8
A-3	Integer Arithmetic Instructions	A-15
A-4	Integer Compare Instructions	A-16
A-5	Integer Logical Instructions	A-16
A-6	Integer Rotate Instructions	A-16
A-7	Integer Shift Instructions	A-17
A-8	Floating-Point Arithmetic Instructions	A-17
A-9	Floating-Point Multiply-Add Instructions	A-18
A-10	Floating-Point Rounding and Conversion Instructions	A-18
A-11	Floating-Point Compare Instructions	A-18
A-12	Floating-Point Status and Control Register Instructions	A-18
A-13	Integer Load Instructions	A-19
A-14	Integer Store Instructions	A-19
A-15	Integer Load and Store with Byte-Reverse Instructions	A-20
A-16	Integer Load and Store Multiple Instructions	A-20
A-17	Integer Load and Store String Instructions	A-20
A-18	Memory Synchronization Instructions	A-21
A-19	Floating-Point Load Instructions	A-21
A-20	Floating-Point Store Instructions	A-21
A-21	Floating-Point Move Instructions	A-22
A-22	Branch Instructions	A-22
A-23	Condition Register Logical Instructions	A-22
A-24	System Linkage Instructions	A-22
A-25	Trap Instructions	A-23
A-26	Processor Control Instructions	A-23
A-27	Cache Management Instructions	A-23
A-29	Lookaside Buffer Management Instructions	A-24

Tables

Table Number	Title	Page Number
A-30	I-Form	A-24
A-31	B-Form	A-24
A-28	Segment Register Manipulation Instructions	A-24
A-32	SC-Form	A-25
A-33	D-Form	A-25
A-34	DS-Form	A-26
A-35	X-Form	A-27
A-36	XL-Form	A-31
A-37	XFX-Form	A-31
A-38	XFL-Form	A-32
A-39	XS-Form	A-32
A-40	XO-Form	A-32
A-41	A-Form	A-33
A-42	M-Form	A-34
A-43	MD-Form	A-34
A-44	MDS-Form	A-34
A-45	PowerPC Instruction Set Legend	A-35
B-1	32-Bit Instructions Not Implemented by the e300 core	B-1
B-2	64-Bit Instructions Not Implemented by the e300 core	B-1
B-3	64-Bit SPR Encoding Not Implemented by the e300 core	B-2

Tables

Table Number	Title	Page Number

About This Book

The primary objective of this reference manual is to describe the functionality of the microprocessors in the e300 core family, which are PowerPC™ microprocessors built on Power Architecture™ technology. The e300 designs are based on the MPC603e microprocessor. This reference manual describes the e300c1, e300c2, and e300c3, configurations. Unless otherwise noted, the information presented here applies to all the e300 cores. The e300c1 has similar functionality to the G2_LE core and any differences in functionality are summarized in Table 1-3. The e300c2, e300c3 have similar functionality to the e300c1 core and any differences regarding functionality are summarized in Section 1.5, "Differences Between e300 Cores." This book is intended as a companion to the *Programming Environments Manual for 32-Bit Implementations of the PowerPC™ Architecture* (referred to as the *Programming Environments Manual*), which describes the features common to PowerPC processors and cores and indicates those features that are optional or that may be implemented differently in the design of each processor and core.

NOTE

About the Companion *Programming Environments Manual*

The PowerPC architecture definition is flexible to support a broad range of processors. Note that the *Programming Environments Manual* describes only architecture features for 32-bit implementations.

Contact your sales representative, or visit the website on the inside cover of this manual for a copy of the *Programming Environments Manual*.

This reference manual and the *Programming Environments Manual* distinguish between the three levels, or programming environments as follows:

- User instruction set architecture (UISA)—The UISA defines the architecture level to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, memory conventions, and the memory and programming models seen by application programmers.
- Virtual environment architecture (VEA)—The VEA, which is the smallest component of the PowerPC architecture, defines additional user-level functionality that falls outside typical user-level software requirements. The VEA describes the memory model for an environment in which multiple processors or other devices can access external memory, and defines aspects of the cache model and cache control instructions from a user-level perspective. The resources defined by the VEA are particularly useful for optimizing memory accesses and for managing resources in an environment in which other processors and devices can access external memory.
- Operating environment architecture (OEA)—The OEA defines supervisor-level resources typically required by an operating system. The OEA defines the memory management model, supervisor-level registers, and interrupt model.

  Implementations that conform to the OEA also conform to the UISA and VEA.

Note that some resources are defined more generally at one level in the architecture and more specifically at another. For example, conditions that cause a floating-point interrupt are defined by the UISA, while the interrupt mechanism itself is defined by the OEA.

For ease in reference, topics in this book are presented in the same order as the *Programming Environments Manual*. Topics build on one another, beginning with a description and complete summary of the e300 core register model and followed by the instruction set model and progressing to more specific, architecture-based topics regarding the cache, interrupt, and memory management models. As such, chapters may include information from multiple levels of the architecture. (For example, the discussion of the cache model uses information from both the VEA and the OEA.)

The PowerPC Architecture: A Specification for a New Family of RISC Processors defines the architecture from the perspective of the three programming environments and remains the defining document for the PowerPC architecture. For information about ordering Freescale documentation, see “Suggested Reading” on page xxv.

The information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. As with any technical documentation, it is the readers’ responsibility to be sure they are using the most recent version of the documentation. For more information, contact your sales representative.

To locate any published errata or updates for this reference manual, refer to the world-wide web at http://www.freescale.com.

A list of major differences between the G2 core, the G2_LE core, and the e300 core configurations are provided in Section 1.4, “Differences Between Cores.” Furthermore, the minor differences between cores are documented by footnotes throughout this book.

Audience

This manual is intended to be used as a reference for many semiconductor products targeting a range of markets including automotive, communication, consumer, networking, and computer peripherals. It is intended for system software and hardware developers and applications programmers who want to develop products using the cores. It is assumed that the reader understands operating systems, core system design, and details of the PowerPC architecture.

Organization

Following is a summary and a brief description of the major sections of this manual:

- Chapter 1, “Overview,” is useful for readers who want a general understanding of e300 features and functions and the differences between the e300 and G2 cores. It generally describes the register set, instruction set and addressing modes, cache model, interrupt model, memory management model, instruction timing, system support interface, and debug features for the e300 core. This chapter indicates which features are defined by the PowerPC architecture and which ones are e300-specific.
- Chapter 2, “Register Model,” provides a brief synopsis of the registers implemented in the e300 core, including architecture-defined and implementation-specific registers.

- Chapter 3, “Instruction Set Model,” provides a brief description of the operand conventions, an overview of addressing modes, and a list of the instructions implemented by the e300 core. Note that instructions are organized by functions.
- Chapter 4, “Instruction and Data Cache Operation,” provides a discussion of the cache and memory model as implemented on the e300 core.
- Chapter 5, “Interrupts and Exceptions,” describes the interrupt model defined in the PowerPC OEA, and the specific interrupt model implemented on the e300 core.
- Chapter 6, “Memory Management,” describes the e300 core’s implementation of the memory management unit specifications provided by the OEA.
- Chapter 7, “Instruction Timing,” provides information about latencies, interlocks, special situations, and various conditions to help make programming more efficient. This chapter is of special interest to software engineers and system designers.
- Chapter 8, “Core Interface Operation,” provides an overview of individual signals of the e300 core. It also provides a description of the coherent system bus (CSB) bus.
- Chapter 9, “Power Management,” provides information about the power saving modes for the e300 core.
- Chapter 10, “Debug Features,” provides information about the debug features of the e300 core. This chapter also describes trace facility debug features for e300 core.
- Appendix A, “Instruction Set Listings,” lists all the PowerPC instructions while indicating those instructions that are not implemented by the e300 core; it also includes the instructions that are specific to the e300 core. Instructions are grouped according to mnemonic, opcode, function, and form. Also included is a quick reference table that contains general information, such as the architecture level, privilege level, and form, and indicates if the instruction is 64-bit and optional.
- Appendix B, “Instructions Not Implemented,” provides a list of the 32- and 64-bit instructions that are not implemented in the e300 core.
- Appendix C, “Revision History,” lists the major differences between revisions of this reference manual.
- This reference manual also includes a glossary and an index.

Suggested Reading

This section lists additional reading that provides background for the information in this reference manual, as well as general information about the PowerPC architecture.

General Information

The following documentation, available through Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA, provides useful information about the PowerPC architecture and computer architecture in general:

- *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.

For updates to the specification, see

http://www-1.ibm.com/support/docview.wss?uid=pub1sa14208300

- *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture*, by Apple Computer, Inc., International Business Machines, Inc., and Motorola, Inc.
- *Computer Architecture: A Quantitative Approach*, Second Edition, John L. Hennessy and David A. Patterson.
- *Computer Organization and Design: The Hardware/Software Interface*, Second Edition, David A. Patterson and John L. Hennessy.
- *Inside Macintosh: PowerPC System Software*, Addison-Wesley Publishing Company, One Jacob Way, Reading, MA, 01867; Tel. (800) 282-2732 (U.S.A.), (800) 637-0029 (Canada), (716) 871-6555 (International).

Related Documentation

Freescale documentation is available from the sources listed on the back cover of this manual; the document order numbers are included in parentheses for ease in ordering:

- *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture* (MPCFPE32B)—Describes resources defined by the PowerPC architecture.
- User's and reference manuals—These books provide details about individual implementations and are intended for use with the *Programming Environments Manual.*
- Addenda/errata to user's or reference manuals—Because some processors have follow-on devices, an addendum is provided that describes the additional features and functionality changes. These addenda are intended for use with the corresponding book.
- *Implementation Variances Relative to Rev. 1 of the Programming Environments Manual* is available at http://www.freescale.com.
- Technical summaries—Each device has a technical summary that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's user's or reference manual.
- Application notes—These short documents contain useful information about specific design issues useful to programmers and engineers working with Freescale processors.

Additional literature is published as new processors become available. For a current list of documentation, refer to: http://www.freescale.com.

Conventions

This document uses the following notational conventions:

mnemonics	Instruction mnemonics are shown in lowercase bold
italics	Italics indicate variable command parameters, for example, **bcctr***x* Book titles in text are set in italics
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number

rA, **r**B	Instruction syntax used to identify a source GPR
rA|0	Contents of a specified GPR or the value 0
rD	Instruction syntax used to identify a destination GPR
frA, **fr**B, **fr**C	Instruction syntax used to identify a source FPR
frD	Instruction syntax used to identify a destination FPR
REG[FIELD]	Abbreviations or acronyms for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.
x	In certain contexts, such as a signal encoding, this indicates a don't care
n	Used to express an undefined numerical value
¬	NOT logical operator
&	AND logical operator
|	OR logical operator
0000	Indicates reserved bits or bit fields in a register. Although these bits may be written to as either ones or zeros, they are always read as zeros.

Acronyms and Abbreviations

Table i contains acronyms and abbreviations that are used in this reference manual.

Table i. Acronyms and Abbreviated Terms

Term	Meaning
ABE	Address bus enable
ALU	Arithmetic logic unit
BAT	Block address translation
BATL	Block address translation lower
BATU	Block address translation upper
BE	Branch trace enable
BIST	Built-in self test
BIU	Bus interface unit
BL	Block size mask
BPU	Branch processing unit
BUID	Bus unit ID
CE	Critical interrupt enable
CIA	Current instruction address
CMOS	Complementary metal-oxide semiconductor
CMP	IABR compare type

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
CMP2	IABR2 compare type
COP	Common on-chip processor
CQ	Completion queue
CR	Condition register
CSRR0	Critical interrupt save/restore register 0
CSRR1	Critical interrupt save/restore register 1
CTR	Count register
DABR	Data address breakpoint register
DABR2	Data address breakpoint register 2
DAR	Data address register
DBAT	Data BAT
DBCR	Data address control register
DCE	Data cache enable
DCFI	Data cache flash invalidate
DCMP	Data TLB compare
DEC	Decrementer register
DLOCK	Data cache lock
DMISS	Data TLB miss address
DMMU	Data memory management unit
DPM	Dynamic power management enable
DR	Data address translation enable
DSISR	Register used for determining the source of a DSI interrupt
DTLB	Data translation lookaside buffer
DWLCK	Data cache way-lock
EA	Effective address
EAR	External access register
ECC	Error checking and correction
EE	External interrupt enable
FE0	Floating-point exception model 0
FE1	Floating-point exception model 1
FIFO	First-in-first-out
FP	Floating-point available
FPR	Floating-point register

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
FPSCR	Floating-point status and control register
FPU	Floating-point unit
GPR	General-purpose register
HBE	High BAT enable
HID0	Hardware implementation register 0
HID1	Hardware implementation register 1
HID2	Hardware implementation register 2
I	Cache-inhibited
IABR	Instruction address breakpoint register 1
IABR2	Instruction address breakpoint register 2
IBAT	Instruction BAT
IBCR	Instruction breakpoint control register
ICE	Instruction cache enable
ICFI	Instruction cache flash invalidate
ICMP	Instruction TLB compare
IEE	External interrupt enable
IEEE	Institute of Electrical and Electronics Engineers
IFEM	Instruction fetch enable M (bit)
ILE	Interrupt little-endian mode
ILOCK	Instruction cache lock
IMISS	Instruction TLB miss address
IMMU	Instruction memory management unit
IP	interrupt prefix
IQ	Instruction queue
IR	Instruction address translation enable
ITLB	Instruction translation lookaside buffer
IU	Integer unit
IWLCK	Instruction cache way lock
L2	Secondary cache
LE	Little-endian mode enable
LET	True little-endian mode bit
LIFO	Last-in-first-out
LR	Link register

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
LRU	Least recently used
lsb	Least-significant bit
LSB	Least-significant byte
LSU	Load/store unit
M	Memory-coherent
MBAR	System memory base address
ME	Machine check enable
MEI	Modified/exclusive/invalid
MESI	Modified/exclusive/shared/invalid—cache coherency protocol
MFG	Manufacturing revision tag
MJREV	Major processor design revision indicator
MMU	Memory management unit
MNREV	Minor processor design revision indicator
MQ	MQ register
msb	Most-significant bit
MSB	Most-significant byte
MSR	Machine state register
NaN	Not a number
NI	Non-IEEE mode bit
No-op	No operation
NOOPTI	No-op the data cache touch instructions
OEA	Operating environment architecture
PID	Processor identification tag
PIR	Processor identification register
PLL	Phase-locked loop
POR	Power-on reset
POW	Power management enable
POWER	Performance optimized with enhanced RISC architecture
PR	Privilege level
PROC	Processor revision tag
PT	Processor ID type tag
PTE	Page table entry
PTEG	Page table entry group

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
PVR	Processor version register
RAW	Read-after-write
RI	Recoverable interrupt
RID	Resource ID
RISC	Reduced instruction set computing
RTL	Register transfer language
RWITM	Read with intent to modify
SDR1	Register that specifies the page table base address for virtual-to-physical address translation
SE	Single-step trace enable
SIG_TYPE	Combinational signal type
SMI	System management interrupt
SOC	System-on-a-chip
SPR	Special-purpose register
SR	Segment register
SRR0	Machine status save/restore register 0
SRR1	Machine status save/restore register 1
SRU	System register unit
SVR	System version register
T	Translation control bit
TAP	Test access port
TB	Time base facility
TBL	Time base lower register
TBU	Time base upper register
TGPR	Temporary GPR remapping
TLB	Translation lookaside buffer
TTL	Transistor-to-transistor logic
UIMM	Unsigned immediate value
UISA	User instruction set architecture
UTLB	Unified translation lookaside buffer
UUT	Unit under test
VEA	Virtual environment architecture
VPN	Virtual page number
W	Write-through

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
WAR	Write-after-read
WAW	Write-after-write
WIMG	Write-through/caching-inhibited/memory-coherency enforced/guarded bits
XATC	Extended address transfer code
XER	Register used for indicating conditions such as carries and overflows for integer operations

Terminology Conventions

Table ii describes terminology conventions used in this manual.

Table ii. Terminology Conventions

The Architecture Specification	This Manual
Data storage interrupt (DSI)	DSI interrupt
Extended mnemonics	Simplified mnemonics
Fixed-point unit (FXU)	Integer unit (IU)
Instruction storage interrupt (ISI)	ISI interrupt
Interrupt	Interrupt
Privileged mode (or privileged state)	Supervisor-level privilege
Problem mode (or problem state)	User-level privilege
Real address	Physical address
Relocation	Translation
Storage (locations)	Memory
Storage (the act of)	Access
Store in	Write back
Store through	Write through

Table iii describes instruction field notation used in this manual.

Table iii. Instruction Field Conventions

The Architecture Specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D (respectively)
BF, BFA	**crf**D, **crf**S (respectively)
D	d
DS	ds
FLM	FM

Table iii. Instruction Field Conventions (continued)

FRA, FRB, FRC, FRT, FRS	**fr**A, **fr**B, **fr**C, **fr**D, **fr**S (respectively)
FXM	CRM
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S (respectively)
SI	SIMM
U	IMM
UI	UIMM
/, //, ///	0...0 (shaded)

Chapter 1
Overview

This chapter provides an overview of features for the embedded microprocessors in the e300 core family, which are PowerPC™ microprocessors built on Power Architecture™ technology. Throughout this document, the terms ‘e300 core’, ‘core’, and ‘processor’ are used interchangeably. The terms ‘e300c1’, ‘e300c2’, and ‘e300c3’ are used when describing an implementation-specific feature or when a difference exists between different configurations. The term ‘e300’ is used when describing a feature that pertains to the family of e300 processors.

1.1 Overview

This section describes the details of the e300 core, provides a block diagram showing the major functional units (see Section 3.1.2, “Endian Modes and Byte Ordering”), and briefly describes how these units interact. All differences between the e300 and previous PowerPC implementations derived from the MPC603e processor are noted. See Section 1.5, “Differences Between e300 Cores for a differences description of the e300 core configurations.

The e300 core is a low-power implementation of this microprocessor family of reduced instruction set computing (RISC) microprocessors. The core implements the 32-bit portion of the PowerPC architecture, which defines 32-bit effective addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 and 64 bits.

The core is a superscalar processor that can issue and retire as many as three instructions per clock cycle. Instructions can execute out of program order for increased performance; however, the core makes completion appear sequential.

The e300 core integrates independent execution units including: an integer unit (IU) a floating-point unit (FPU), a branch processing unit (BPU), a load/store unit (LSU), and a system register unit (SRU). The e300c2 and e300c3 integrate an additional integer unit for a total of two IUs. Note that the e300c2 does not include an FPU. The ability to execute instructions in parallel and the use of simple instructions with rapid execution times yield high efficiency and throughput for e300-core-based systems. Most integer instructions execute in one clock cycle. The additional IUs along with enhanced multipliers in the e300c2 and e300c3 improve multiply instructions to a maximum two-cycle latency, a significant improvement from previous processors. In the e300c1 and e300c3 core, the FPU is pipelined so a single-precision multiply-add instruction can be issued and completed every clock cycle. The e300c1 and e300c3 core provides hardware support for all single- and double-precision floating-point operations for most value representations and all rounding modes.

Figure 1-1 shows a block diagram of the e300c1 core.

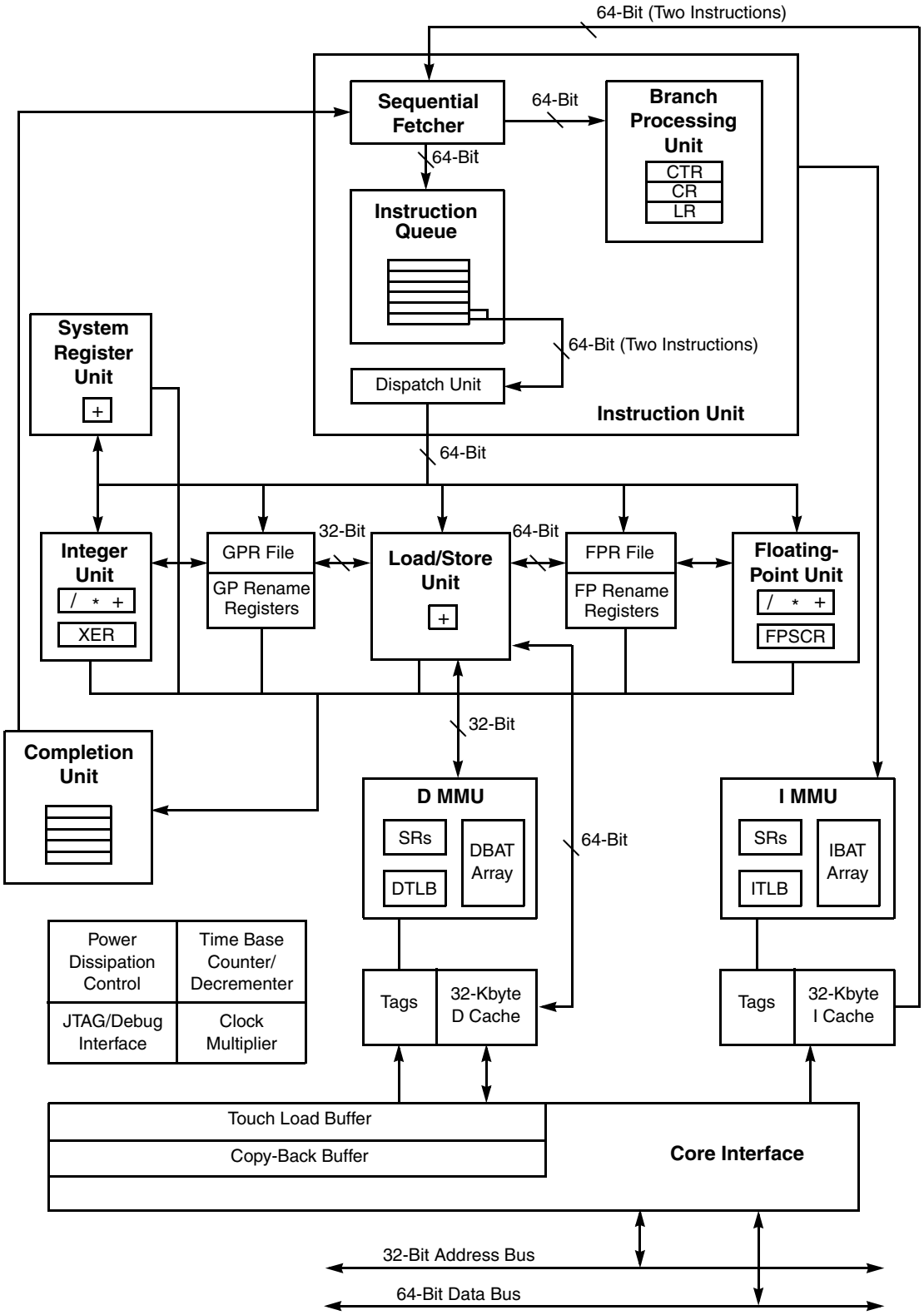


Figure 1-1. e300c1 Core Block Diagram

Figure 1-2 is a block diagram of the e300c2 core. Note that it does not support floating-point operations.

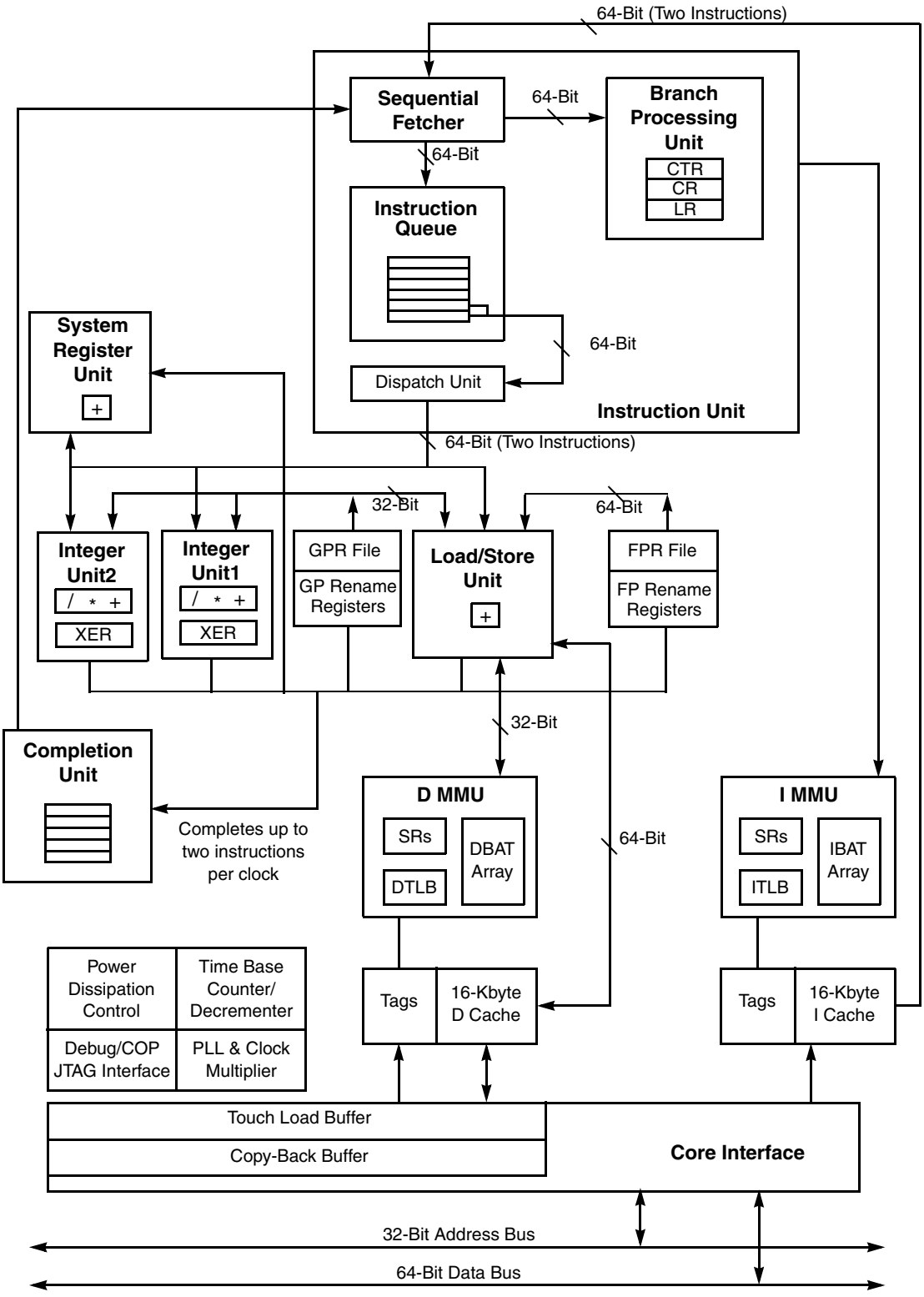


Figure 1-2. e300c2 Core Block Diagram

Figure 1-3 shows a block diagram of the e300c3 core. Note that the e300c3 supports floating-point operations and includes two integer units.

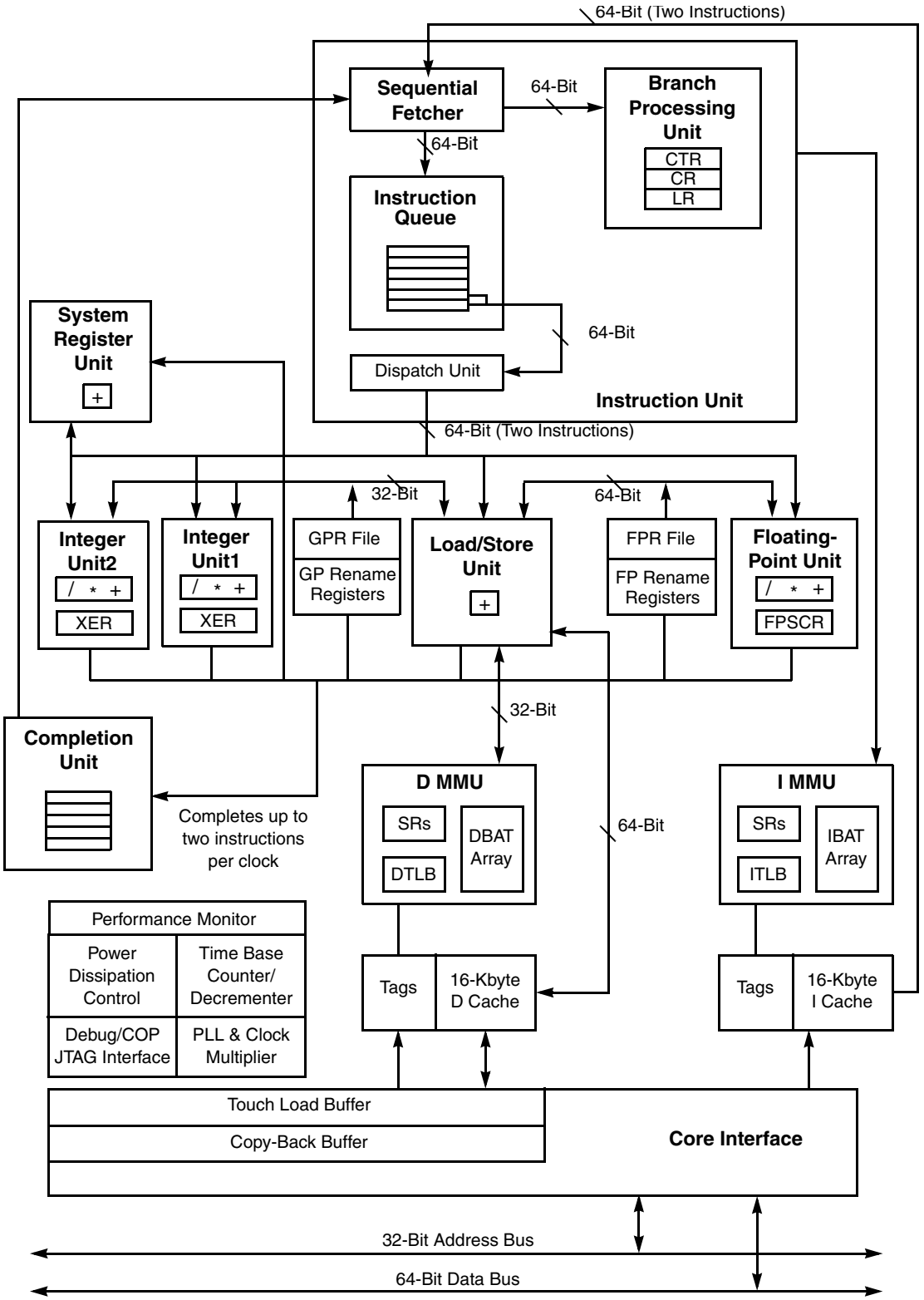


Figure 1-3. e300c3 Core Block Diagram

The e300c1 provide independent, on-chip, 32-Kbyte, eight-way, set-associative, physically-addressed caches for instructions and data, and on-chip instruction and data memory management units (MMUs).The e300c2 and e300c3 include 16-Kbyte, four way set-associative instruction and data caches. The MMUs contain 64-entry, two-way, set-associative, data and instruction translation lookaside buffers (DTLB and ITLB) that provide support for demand-paged, virtual-memory, address translation and variable-sized block translation. The TLBs use a least recently used (LRU) replacement algorithm and the caches use a pseudo least recently used algorithm (PLRU).

The core also supports block address translation through the use of two independent instruction and data block address translation (IBAT and DBAT) arrays, each containing eight pairs of BATs, an increase from four pairs of each type of BATs in the G2 core. This increase provides more flexibility in protecting accesses and providing translation on a segment, block, or page basis for memory accesses and I/O accesses. Effective addresses are compared simultaneously with all eight entries in the BAT array during block translation. In accordance with the PowerPC architecture, if an effective address hits in both the TLB and BAT array, the BAT translation takes priority.

As part of the coherent system bus (CSB), the e300 core has a 64-bit data bus and a 32-bit address bus. During normal operation, the e300 core provides a three-state (modified, exclusive, and invalid) coherency protocol which is a compatible subset of a four-state (modified/exclusive/shared/invalid) MESI protocol. However, the e300 data cache contains a programmable MESI extension that supports the shared cache coherency state (similar to other PowerPC processors). Both protocols operate coherently in systems that contain four-state caches. The core also supports single-beat and burst data transfers for memory accesses and supports memory-mapped I/O operations.

The true little-endian mode is another enhanced capability of the e300 core. Unlike the PowerPC little-endian mode (which manipulates only the address bits), no longer supported on the e300, the true little-endian mode actually operates on true little-endian instructions and data from memory.

The critical interrupt is an additional interrupt in the e300 core and has higher priority order than the system management interrupt. Also, debug features are improved in the e300. Additional SPRG interrupt handling registers are provided for enhancing flexibility for the operating system.

The e300c3 include a performance monitor facility that provides the ability to monitor and count predefined events such as core clocks, misses in the instruction cache, data cache, or L2 cache, types of instructions dispatched, mispredicted branches, and other occurrences. The count of such events (which may be an approximation) can be used to trigger the performance monitor interrupt. Section 1.1.7.5, “Core Performance Monitor,” describes the operation of the performance monitor diagnostic tool. This functionality is fully described in Chapter 11, “Performance Monitor.”

1.1.1 Features

This section describes the major features of the e300 core:

- High-performance, superscalar microprocessor core
  - As many as three instructions issued and retired per clock (two instructions plus one branch instruction)
  - As many as five instructions in execution per clock
  - Single-cycle execution for most instructions

  - — Pipelined floating-point unit (FPU) for all single-precision and double-precision operations (not included in the e300c2)
- Independent execution units and two register files
  - — Branch processing unit (BPU) featuring static branch prediction
  - — Two 32-bit integer units (IU) in the e300c2 and e300c3. One 32-bit integer units (IU) in the e300c1.
  - — FPU based on the **IEEE**® 754™ standard for both single- and double-precision operations
  - — Load/store unit (LSU) for data transfer between data-cache and general-purpose registers (GPRs) and floating-point registers (FPRs)
  - — System register unit (SRU) that executes condition register (CR), special-purpose register (SPR), and integer add/compare instructions. Add/compare instructions are also executed in the IUs.
  - — Thirty-two 32-bit GPRs for integer operands
  - — Thirty-two 64-bit FPRs for single- or double-precision operands
- High instruction and data throughput
  - — Zero-cycle branch capability (branch folding)
  - — Programmable static branch prediction on unresolved conditional branches
  - — Two integer units with enhanced multipliers in the e300c2 and e300c3 for increased integer instruction throughput and a maximum two-cycle latency for multiply instructions
  - — Instruction fetch unit capable of fetching two instructions per clock from the instruction cache
  - — A six-entry instruction queue (IQ) that provides lookahead capability
  - — Independent pipelines with feed-forwarding that reduces data dependencies in hardware
  - — 32-Kbyte data cache and 32-Kbyte instruction cache with parity—eight-way, set-associative, physically addressed, PLRU replacement algorithm on the e300c1. 16-Kbyte, four-way set-associative instruction and data caches on the e300c2 and e300c3.
  - — Cache write-back or write-through operation programmable on a per-page or per-block basis
  - — Features for instruction and data cache locking and protection
  - — BPU that performs CR lookahead operations
  - — Address translation facilities for 4-Kbyte page size, variable block size, and 256-Mbyte segment size
  - — A 64-entry, two-way, set-associative ITLB and DTLB
  - — Eight-entry data and instruction BAT arrays providing 128-Kbyte to 256-Mbyte blocks
  - — Software table search operations and updates supported through fast trap mechanism
  - — 52-bit virtual address; 32-bit physical address
- Facilities for enhanced system performance
  - — A 64-bit split-transaction internal data bus interface to the coherent system bus (CSB) with burst transfers
  - — Support for one-level address pipelining on the CSB interface
  - — True little-endian mode for compatibility with other true little-endian devices

  - Critical interrupt support
  - Hardware support for misaligned little-endian accesses
- Integrated power management
  - Internal processor/bus clock multiplier ratios
  - Three power-saving modes: doze, nap, and sleep
  - Automatic dynamic power reduction when internal functional units are idle
- In-system testability and debugging features through JTAG boundary-scan capability

Features specific to the e300 core not present on the G2 processors follow:

- Enhancements to the register set
  - The e300 core has one more HID0 bit than the G2:
    - The enable cache parity checking (ECPE) bit, HID0[1], gives the e300 core the ability to enable the taking of a machine check interrupt based on the detection of a cache parity error
- Enhancements to cache implementation
  - 32-Kbyte, eight-way, set-associative instruction and data cache on the e300c1; 16-Kbyte, four-way set-associative instruction and data cache on the e300c2 and e300c3
  - Full parity checking is performed on both instruction and data cache memory arrays
  - Lockable L1 instruction and data caches—entire cache or on a per-way basis up to 7 of 8 ways on the e300c1 and 3 of 4 ways on the e300c2 and e300c3
  - New **icbt** instruction supports initialization of instruction cache
  - Data cache supports four-state MESI coherency protocol
  - The instruction cache is blocked only until the critical load completes (hit under reloads allowed)
  - Instruction cancel mechanism improves utilization of instruction cache by supporting hits-under-cancels and misses-under-cancels.
  - The critical double word is simultaneously written to the cache and forwarded to the requesting unit, thus minimizing stalls due to load delays.
  - Data cache queue sharing makes cast-outs and snoop pushes more efficient
  - Provides for an optional data cache operation broadcast feature (enabled by HID0[ABE]) that allows for coherent system management. All of the data cache control instructions, except **dcbz** (**dcbi**, **dcbf**, and **dcbst**) require that HID0[ABE] be enabled to broadcast.
  - Instruction fetch burst feature allows all instruction fetches from caching-inhibited space to be performed on the bus as burst transactions
- Interrupts
  - The e300 core offers hardware support for misaligned little-endian accesses. Little-endian load/store accesses that are not on a word boundary, except for strings and multiples, generate interrupts under the same circumstances as big-endian accesses.
  - The e300 core supports true little-endian mode to minimize the impact on software porting from true little-endian systems.

  - An input interrupt signal, *cint*, is provided to trigger the critical interrupt exception on the e300 core. Also, an interrupt *pm_event_int* is provided on the e300c3 to support the performance monitor.
- Bus clock—PLL configuration signals include seven signals for settings and control: *pll_cfg[0:6]*.
- Debug features
  - Breakpoint status recorded in DBCR and IBCR control registers
  - Two signals for the debug interface: *stopped* and *ext_halt*
  - Performance monitor registers for system analysis in the e300c3

Figure 1-1, Figure 1-2, and Figure 1-3 provide block diagrams of the e300 cores that show how the execution units—IU, FPU, BPU, LSU, and SRU—operate independently and in parallel. It should be noted that this is a conceptual diagram and does not attempt to show how these features are physically implemented on the device.

The e300 core provides address translation and protection facilities, including an ITLB, DTLB, and instruction and data BAT arrays. Instruction fetching and issuing are handled in the instruction unit. Translation of addresses for cache or external memory accesses are handled by the MMUs. Both units are discussed in more detail in Section 1.1.2, "Instruction Unit," and Section 1.1.5.1, "Memory Management Units (MMUs)."

1.1.2 Instruction Unit

As shown in Figure 1-1, Figure 1-2, and Figure 1-3, the e300 core instruction unit, containing a fetch unit, instruction queue, dispatch unit, and BPU, provides centralized control of instruction flow to the execution units. The instruction unit determines the address of the next instruction to be fetched based on information from the sequential fetcher and from the BPU.

The instruction unit fetches the instructions from the instruction cache into the instruction queue. The BPU receives branch instructions from the fetcher and uses static branch prediction to allow fetching from a predicted instruction stream while a conditional branch is evaluated. The BPU folds out for unconditional branch instructions and conditional branch instructions unaffected by instructions in the execution pipeline.

Instructions issued beyond a predicted branch cannot complete execution until the branch is resolved, preserving the programming model of sequential execution. If any of these are branch instructions, they are decoded but not issued. Instructions to be executed by the FPU, IU, LSU, and SRU are issued and allowed to progress up to the register write-back stage. Write-back is allowed when a correctly predicted branch is resolved, and execution continues along the predicted path.

If branch prediction is incorrect, the instruction unit flushes all predicted path instructions, and instructions are issued from the correct path.

1.1.2.1 Instruction Queue and Dispatch Unit

The instruction queue (IQ), shown in Figure 1-1, Figure 1-2, and Figure 1-3, holds as many as six instructions and loads up to two instructions from the instruction unit during a single cycle. The instruction fetch unit continuously loads as many instructions as space in the IQ allows. Instructions are dispatched to

their respective execution units from the dispatch unit at a maximum rate of two instructions per cycle. Dispatching is facilitated to the IUs, FPU, LSU, and SRU by the provision of a reservation station at each unit. The dispatch unit performs source and destination register dependency checking, determines dispatch serializations, and inhibits subsequent instruction dispatching as required.

For a more detailed overview of instruction dispatch, see Section 1.3.6, "Instruction Timing."

1.1.2.2 Branch Processing Unit (BPU)

The BPU receives branch instructions from the fetch unit and performs CR lookahead operations on conditional branches to resolve them early, achieving the effect of a zero-cycle branch in many cases.

The BPU uses a bit in the instruction encoding to predict the direction of the conditional branch. Therefore, when an unresolved conditional branch instruction is encountered, the core fetches instructions from the predicted target stream until the conditional branch is resolved.

The BPU contains an adder to compute branch target addresses and three user-control registers: the link register (LR), the count register (CTR), and the conditional register (CR). The BPU calculates the return pointer for sub-routine calls and saves it into the LR for certain types of branch instructions. The LR also contains the branch target address for the Branch Conditional to Link Register (**bclr***x*) instruction. The CTR contains the branch target address for the Branch Conditional to Count Register (**bcctr***x*) instruction. The contents of the LR and CTR can be copied to or from any GPR. Because the BPU uses dedicated registers rather than GPRs or FPRs, execution of branch instructions is largely independent from execution of integer and floating-point instructions.

1.1.3 Independent Execution Units

The PowerPC architecture's support for independent execution units allows implementation of processors with out-of-order instruction execution. For example, because branch instructions do not depend on GPRs or FPRs, branches can often be resolved early, eliminating stalls caused by taken branches.

The four other execution units and the completion unit are described in the following sections.

1.1.3.1 Integer Unit (IU)

The IU executes all integer instructions. The IU executes one integer instruction at a time, performing computations with its arithmetic logic unit (ALU), multiplier, divider, and XER register. Most integer instructions are single-cycle instructions. The 32 GPRs hold integer operands. Stalls due to contention for GPRs are minimized by the automatic allocation of rename registers. The core writes the contents of the rename registers to the appropriate GPR when integer instructions are retired by the completion unit. The e300c2 and e300c3 provide two integer units for greater integer instruction throughput along with enhanced multipliers in each IU for faster multiply-instruction execution.

1.1.3.2 Floating-Point Unit (FPU)

The FPU contains a single-precision multiply-add array and the floating-point status and control register (FPSCR). The multiply-add array allows the core to efficiently implement multiply and multiply-add operations. The FPU is pipelined so that single- and double-precision instructions can be issued

back-to-back. The 32 FPRs are provided to support floating-point operations. Stalls due to contention for FPRs are minimized by the automatic allocation of rename registers. The core writes the contents of the rename registers to the appropriate FPR when floating-point instructions are retired by the completion unit. The e300c2 does not include an FPU and does not support floating-point operations.

The e300c1 and e300c3 core support all floating-point data types based on the **IEEE** 754 standard (normalized, denormalized, NaN, zero, and infinity) in hardware, eliminating the latency incurred by software interrupt routines.

1.1.3.3 Load/Store Unit (LSU)

The LSU executes all load and store instructions and provides the data transfer interface between the GPRs, FPRs, and the cache/memory subsystem. The LSU calculates effective addresses, performs data alignment, and provides sequencing for load/store string and multiple instructions.

Load and store instructions are issued and executed in program order; however, the memory accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering.

Cacheable loads, when free of data bus dependencies, can execute out of order with a maximum throughput of one per cycle and with a two-cycle total latency. Data returned from the cache is held in a rename register until the completion logic commits the value to a GPR or FPR. Stores cannot be executed in a predicted manner and are held in the store queue until the completion logic signals that the store operation is to be completed to memory. The core executes store instructions with a maximum throughput of one per cycle and with a three-cycle total latency. The time required to perform the actual load or store depends on whether the operation involves the cache, system memory, or an I/O device.

1.1.3.4 System Register Unit (SRU)

The SRU executes various system-level instructions, including condition register logical operations and move to/from special-purpose register instructions. It also executes integer add/compare instructions. In order to maintain system state, most instructions executed by the SRU are completion-serialized; that is, the instruction is held for execution in the SRU until all prior instructions issued have completed. Results from completion-serialized instructions executed by the SRU are not available or forwarded for subsequent instructions until they complete.

1.1.4 Completion Unit

The completion unit tracks instructions in program order from dispatch through execution and then completes. Completing an instruction commits the core to any architectural register changes caused by that instruction. In-order completion ensures the correct architectural state when the core must recover from a mispredicted branch or an interrupt.

Instruction state and other information required for completion is kept in a five-entry FIFO completion queue. A single completion queue entry is allocated for each instruction once it enters the execution unit from the dispatch unit. An available completion queue entry is a required resource for dispatch; if no completion entry is available, dispatch stalls. A maximum of two instructions per cycle are completed in order from the queue.

1.1.5 Memory Subsystem Support

The core provides separate instruction and data caches and MMUs. The core also provides an efficient processor bus interface to facilitate access to main memory and other bus subsystems. The memory subsystem support functions are described in the following sections.

1.1.5.1 Memory Management Units (MMUs)

The core MMUs support up to 4 Petabytes (2^{52}) of virtual memory and 4 Gigabytes (2^{32}) of physical memory (referred to as real memory in the architecture specification) for instruction and data. The MMUs also control access privileges for these spaces on block and page granularities. Referenced and changed status is maintained by the processor for each page to assist implementation of a demand-paged virtual memory system. Note that software assistance is required for the device to maintain reference and changed status. A key bit is implemented to provide information about memory protection violations prior to page table search operations.

The LSU calculates effective addresses for data loads and stores, performs data alignment to and from cache memory, and provides the sequencing for load and store string and multiple word instructions. The instruction unit calculates effective addresses for instruction fetching.

After an EA is generated, its higher-order bits are translated by the appropriate MMU into physical address bits. The lower-order EA bits are the same on the physical address which are directed to the on-chip cache and formed the index into a four-way set-associative tag array. After translating the address, the MMU passes the higher-order physical address bits to the cache and the cache lookup completes. For caching-inhibited accesses or accesses that miss in the cache, the untranslated lower-order address bits are concatenated with the translated higher-order address bits; the resulting 32-bit physical address is then used by the memory unit and the core interface to access external memory.

The MMU also directs the address translation and enforces the protection hierarchy programmed by the operating system in relation to the supervisor/user privilege level of the access and in relation to whether the access is a load or store.

For instruction fetches, the IMMU looks for the address in the ITLB and in the IBAT array. If an address hits both, the IBAT array translation is used. Data accesses cause a lookup in the DTLB and DBAT array. In most cases, the translation is in a TLB and the physical address bits are available to the on-chip cache.

The e300 core implements four more IBAT and four more DBAT entries than the G2.

When the EA misses in the TLBs, the core provides hardware assistance for software to perform a search of the translation tables in memory. The hardware assist consists of the following features:

- Automatic storage of the missed effective address in IMISS and DMISS
- Automatic generation of the primary and secondary hashed real addresses of the page-table entry group (PTEG), which are readable from the HASH1 and HASH2 register locations.

  The HASH data is generated from the contents of the IMISS or DMISS register. The register that is selected depends on the miss (instruction or data) that was last acknowledged.
- Automatic generation of the first word of the page table entry (PTE) of the tables being searched
- A real page address (RPA) register that matches the format of the lower word of the PTE

- TLB access instructions (**tlbli** and **tlbld**) that are used to load an address translation into the instruction or data TLBs
- Shadow registers for GPR0–GPR3 that allow miss code to execute without corrupting the state of any of the existing GPRs. Shadow registers are used only for servicing a TLB miss.

See Section 1.3.5.2, "Implementation-Specific Memory Management," for more information about memory management for the core.

1.1.5.2 Cache Units

The e300c1 provide independent, 32-Kbyte, eight-way, set-associative, instruction and data caches. The e300c2 and e300c3 provide 16-Kbyte, four-way set-associative instruction and data caches. The cache block is 32 bytes long. The caches adhere to a write-back policy, but the e300 core allows control of cacheability, write policy, and memory coherency at the page and block levels. The caches use a pseudo LRU replacement policy.

As shown in Figure 1-1, Figure 1-2, and Figure 1-3, the caches provide a 64-bit interface to the instruction fetch unit and LSU. The surrounding logic selects, organizes, and forwards the requested information to the requesting unit. Write operations to the cache can be performed on a byte basis, and a complete read-modify-write operation to the cache can occur in each cycle.

The load/store and instruction fetch units provide the caches with the address of the data or instruction to be fetched. In the case of a cache hit, the cache returns two words to the requesting unit.

Because the data cache tags are single-ported, simultaneous load/store and snoop accesses cause resource contention. Snoop accesses have the highest priority and are given first access to the tags, unless the snoop access coincides with a tag write; in this case the snoop is retried and must rearbitrate for cache access. Loads or stores deferred due to snoop accesses are performed on the clock cycle following the snoop.

The e300 core includes a new instruction cancel extension. The instruction cancel extension improves utilization of the instruction cache during cancel operations. It allows a new instruction fetch to be issued to the cache or to the bus if a canceled instruction fetch is pending or active on the bus. This supports hit-under-cancel and miss-under-cancel instruction fetch operations.

1.1.6 Bus Interface Unit (BIU)

Because the caches are on-chip, write-back caches, the most common transactions are burst-read memory operations, burst-write memory operations, and single-beat (noncacheable or write-through) memory read and write operations. There can also be address-only operations, variants of the burst and single-beat operations, (for example, global memory operations that are snooped and atomic memory operations), and address retry activity (for example, when a snooped read access hits a modified cache block).

Memory accesses can occur in single-beat (1–8 bytes) and four-beat burst (32 bytes) data transfers on the 64-bit data bus. The address and data buses operate independently to support pipelining and split transactions during memory accesses.

The e300 bus interface unit (BIU) has been enhanced to allow a pipeline slot to become available once a previous transaction has been granted the data bus (that is, as early as when the data tenure starts rather

than after the data tenure completes), thus allowing for greater bus utilization in systems that support it. This is sometimes referred to as 1 1/2-level pipelining.

Typically, memory accesses are weakly ordered, meaning that sequences of operations, including load/store string and multiple instructions, do not necessarily complete in the order they begin. This weak ordering maximizes the efficiency of the bus without sacrificing coherency of the data. The core allows read operations to precede store operations (except when a dependency exists, or in cases where a noncacheable access is performed), and provides support for a write operation to proceed a previously queued read data tenure (for example, allowing a snoop push to be enveloped by the address and data tenures of a read operation). Because the processor can dynamically optimize run-time ordering of load/store traffic, overall performance is improved.

1.1.7 System Support Functions

The e300 core implements several support functions that include power management, time base/decrementer registers for system timing tasks, a JTAG (based on IEEE 1149.1™) interface, hardware debug, and a phase-locked loop (PLL) clock multiplier. These system support functions are described in the following sections.

1.1.7.1 Power Management

The e300 core provides four power modes, selectable by setting the appropriate control bits in the machine state register (MSR) and the hardware implementation register 0 (HID0). When entering into a power mode other than full-power, the core will request entry via a *qreq* signal and will only enter another power mode after an acknowledge ($\overline{qack}$) is received. The four power modes are as follows:

- Full-power—This is the default power state of the e300 core. The e300 core is fully powered and the internal functional units are operating at the full processor clock speed. If the dynamic power management mode is enabled, functional units that are idle will automatically enter a low-power state without affecting performance, software execution, or external hardware.
- Doze—All the functional units of the e300 core are disabled except for the time base/decrementer registers and the bus snooping logic. When the processor is in doze mode, an external asynchronous interrupt, system management interrupt, decrementer interrupt, hard or soft reset, or machine check brings the e300 core into the full-power state. The core in doze mode maintains the PLL in a fully-powered state and locked to the system external clock input (*sysclk*), so a transition to the full-power state takes only a few processor clock cycles.
- Nap—The nap mode further reduces power consumption by disabling bus snooping, leaving only the time base register and the PLL in a powered state. The core returns to the full-power state on receipt of an external asynchronous interrupt, system management interrupt, decrementer interrupt, hard or soft reset, or machine check input ($\overline{mcp}$) signal. A return to full-power state from a nap state takes only a few processor clock cycles.
- Sleep—Sleep mode reduces power consumption to a minimum by disabling all internal functional units; then external system logic may disable the PLL and *sysclk*. Returning the core to the full-power state requires the enabling of the PLL and *sysclk*, followed by the assertion of an external asynchronous interrupt, system management interrupt, hard or soft reset, or $\overline{mcp}$ signal after the time required to relock the PLL.

1.1.7.2 Time Base/Decrementer

The time base is a 64-bit register (accessed as two 32-bit registers) that is incremented once every four bus clock cycles; external control of the time base is provided through the time base enable (*tben*) signal. The decrementer is a 32-bit register that generates a decrementer interrupt after a programmable delay. The contents of the decrementer register are decremented once every four bus clock cycles, and the decrementer interrupt is generated as the count passes through zero.

1.1.7.3 JTAG Test and Debug Interface

The core provides JTAG and hardware debug functions for facilitating board testing and chip debugging. The JTAG test interface (based on IEEE 1149.1) provides a means for boundary-scan testing of the core and the attached system logic. The hardware debug function accesses the JTAG test port, providing a means for executing test routines and facilitating chip and software debugging.

All instruction and data address breakpoints are accessible in the IBCR and DBCR. See Section 1.3.8, “Debug Features,” for more information.

The *stopped* signal allows observation of the internal clock state, and *ext_halt* can be used to force the core into a halted state. See Chapter 5, “Interrupts and Exceptions,” for more information on test signals.

1.1.7.4 Clock Multiplier

The internal clocking of the e300 core is generated from and synchronized to the external clock signal, *sysclk*, by means of a voltage-controlled, oscillator-based PLL. The PLL provides programmable internal processor clock multiplier ratios which multiply the externally supplied clock frequency. The bus clock is the same frequency and is synchronous with *sysclk*. The configuration of the PLL can be read by software from the hardware implementation register 1 (HID1).

1.1.7.5 Core Performance Monitor

The performance monitor provides the ability to count predefined events and processor clocks associated with particular operations, such as cache misses, mispredicted branches, or the number of cycles an execution unit stalls. The count of such events can be used to trigger the performance monitor interrupt.

The performance monitor can be used to do the following:

- Improve system performance by monitoring software execution and then recoding algorithms for more efficiency. For example, memory hierarchy behavior can be monitored and analyzed to optimize task scheduling or data distribution algorithms.
- Characterize processors in environments not easily characterized by benchmarking.
- Help system developers bring up and debug their systems.

The performance monitor uses the following resources:

- The performance monitor mark bit in the MSR (MSR[PMM]). This bit controls which programs are monitored.
- The move to/from performance monitor registers (PMR) instructions, **mtpmr** and **mfpmr**.
- The external input, *pm_event_in*.
- PMRs:

  - The performance monitor counter registers (PMC0–PMC3) are 32-bit counters used to count software-selectable events. Each counter counts up to 128 events. UPMC0–UPMC3 provide user-level read access to these registers. They are identified in Table 1-4.
  - The performance monitor global control register (PMGC0) controls the counting of performance monitor events. It takes priority over all other performance monitor control registers. UPMGC0 provides user-level read access to PMGC0.
  - The performance monitor local control registers (PMLCa0–PMLCa3) control each individual performance monitor counter. Each counter has a corresponding PMLCa register. UPMLCa0–UPMLCa3 provide user-level read access to PMLCa0–PMLCa3).
- The performance monitor interrupt is assigned to interrupt vector 0x0F00.

Software communication with the performance monitor is achieved through PMRs rather than SPRs. The PMRs are used for enabling conditions that can trigger the performance monitor interrupt.

1.2 PowerPC Architecture Implementation

The PowerPC architecture consists of the following layers, and adherence to the PowerPC architecture can be measured in terms of which of the following levels of the architecture is implemented:

- User instruction set architecture (UISA)—Defines the base user-level instruction set, user-level registers, data types, floating-point interrupt model, memory models for a uniprocessor environment, and programming model for a uniprocessor environment.
- Virtual environment architecture (VEA)—Describes the memory model for a multiprocessor environment, defines cache control instructions, and describes other aspects of virtual environments. Implementations that conform to the VEA also adhere to the UISA but may not necessarily adhere to the OEA.
- Operating environment architecture (OEA)—Defines the memory management model, supervisor-level registers, synchronization requirements, and interrupt model. Implementations that conform to the OEA also adhere to the UISA and VEA.

The PowerPC architecture allows a wide range of designs for such features as cache and core interface implementations.

1.3 Implementation-Specific Information

This section describes the PowerPC architecture in general and specific details about the implementation of the e300 core as a low-power, 32-bit member of this PowerPC core family. The main topics addressed are as follows:

- Section 1.3.1, “Register Model,” describes the registers for the operating environment architecture common among e300 cores that implement the PowerPC architecture and describes the programming model. It also describes the additional registers that are unique to the core.
- Section 1.3.2, “Instruction Set and Addressing Modes,” describes the PowerPC instruction set and addressing modes for the OEA, and defines and describes the instructions implemented in the core.
- Section 1.3.3, “Cache Implementation,” describes the cache model that is defined generally for cores that implement the PowerPC architecture by the VEA. It also provides specific details about the e300 core cache implementation.

- Section 1.3.4, “Interrupt Model,” describes the interrupt model of the OEA and the differences in the core interrupt model.
- Section 1.3.5, “Memory Management,” describes generally the conventions for memory management among these cores. This section also describes the core implementation of the 32-bit PowerPC memory management specification.
- Section 1.3.6, “Instruction Timing,” provides a general description of the instruction timing provided by the superscalar, parallel execution supported by the PowerPC architecture and the e300 core.
- Section 1.1.6, “Bus Interface Unit (BIU),” describes the signals implemented on the core.

The e300 core is a high-performance, superscalar processor core. The PowerPC architecture allows optimizing compilers to schedule instructions to maximize performance through efficient use of the PowerPC instruction set and register model. The multiple, independent execution units allow compilers to optimize instruction throughput. Compilers that take advantage of the flexibility of the PowerPC architecture can additionally optimize system performance.

The following sections summarize the features of the core, including both those that are defined by the architecture and those that are unique to the various core implementations.

Specific features of the core are listed in Section 1.1.1, “Features.”

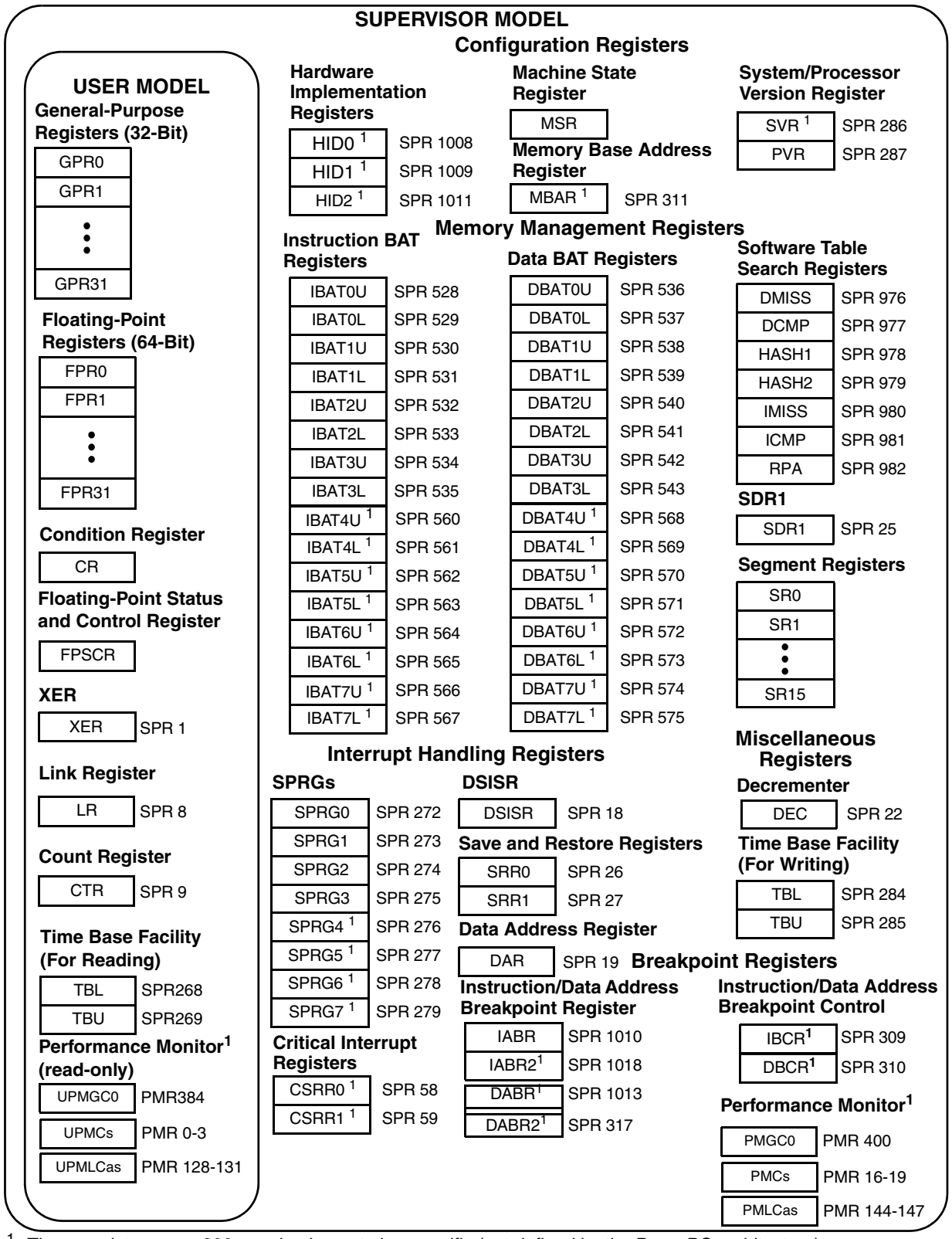
1.3.1 Register Model

The PowerPC architecture defines register-to-register operations for most computational instructions. Source operands for these instructions are accessed from the registers or are provided as immediate values embedded in the instruction opcode. The three-register instruction format allows specification of a target register distinct from the two-source operands. Load and store instructions transfer data between registers and memory.

The e300 core has two levels of privilege: supervisor mode of operation (typically used by the operating system) and user mode of operation (used by the application software). The programming models incorporate 32 GPRs, 32 FPRs, special-purpose registers (SPRs), and several miscellaneous registers. Each core also has its own unique set of hardware implementation (HID) registers.

Having access to privileged instructions, registers, and other resources allows the operating system to control the application environment (providing virtual memory and protecting operating system and critical machine resources). Instructions that control the state of the e300 core, the address translation mechanism, and supervisor registers can be executed only when the core is operating in supervisor mode.

Figure 1-4 shows all the core registers available at the user and supervisor level. The numbers to the right of the SPRs indicate the number that is used in the syntax of the instruction operands for the move to/from SPR instructions.



1 These registers are e300 core implementation-specific (not defined by the PowerPC architecture).

Figure 1-4. e300 Programming Model—Registers

The following sections describe the e300-core-implementation-specific features as they apply to registers.

1.3.1.1 UISA Registers

UISA registers are user-level registers that include the following.

1.3.1.1.1 General-Purpose Registers (GPRs)

The PowerPC architecture defines 32 user-level GPRs that are 32 bits wide in 32-bit cores. The GPRs serve as the data source or destination for all integer instructions.

1.3.1.1.2 Floating-Point Registers (FPRs)

The PowerPC architecture also defines 32 user-level, 64-bit FPRs. The FPRs serve as the data source or destination for floating-point instructions. These registers can contain data objects of either single- or double-precision floating-point formats. FPRs are not included in the e300c2 core.

1.3.1.1.3 Condition Register (CR)

The CR is a 32-bit user-level register that provides a mechanism for testing and branching. It consists of eight 4-bit fields that reflect the results of certain operations, such as move, integer and floating-point comparisons, arithmetic, and logical operations.

1.3.1.1.4 Floating-Point Status and Control Register (FPSCR)

The user-level FPSCR contains all floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the **IEEE** 754 standard. FPSCRs are not included in the e300c2 core.

1.3.1.1.5 User-Level SPRs

The PowerPC architecture defines numerous special purpose registers that serve a variety of functions, such as providing controls, indicating status, configuring the core, and performing special operations. During normal execution, a program can access the registers, as shown in Figure 1-4, depending on the program's access privilege (supervisor or user, determined by the privilege-level bit, MSR[PR]). Note that GPRs and FPRs are accessed through operands that are part of the instructions. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions) or implicit, as the part of the execution of an instruction. Some registers are accessed both explicitly and implicitly. In the e300 core, all SPRs are 32 bits wide.

The following SPRs are accessible by user-level software:

- Link register (LR)—The LR can be used to provide the branch target address and to hold the return address after branch and link instructions. The LR is 32 bits wide in 32-bit implementations.
- Count register (CTR)—The CTR is decremented and tested automatically as a result of branch-and-count instructions. The CTR is 32 bits wide in 32-bit implementations.

- XER register—The 32-bit XER contains the summary overflow bit, integer carry bit, overflow bit, and a field specifying the number of bytes to be transferred by a Load String Word Indexed (**lswx**) or Store String Word Indexed (**stswx**) instruction.

1.3.1.2 VEA Registers

The VEA introduces the time base facility (TB) for reading. The TB is a 64-bit register pair whose contents are incremented once every four core input clock cycles. The TB consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). Note that the time base registers are read-only in user state.

1.3.1.3 OEA Registers

OEA registers are supervisor-level registers that include the following.

1.3.1.3.1 Machine State Register (MSR)

The MSR is a supervisor-level register that defines the state of the core. The contents of this register are saved when an interrupt is taken, and restored when the interrupt handling completes. A critical interrupt interrupt is taken in the e300 core when the $\overline{cint}$ signal is asserted and MSR[CE] is set. The e300 core implements the MSR as a 32-bit register.

1.3.1.3.2 Segment Registers (SRs)

For memory management, 32-bit processors implement sixteen 32-bit SRs. To speed access, the core implements the SRs as two arrays: a main array, for data memory accesses, and a shadow array, for instruction memory accesses. Loading a segment entry with the Move to Segment Register (**mtsr**) instruction loads both arrays.

1.3.1.3.3 Supervisor-Level SPRs

The e300 core, like the G2_LE core, has additional supervisor-level SPRs, which are shown in Figure 1-4. Two critical interrupt SPRs (CSRR0 and CSRR1), eight SPRGs (SPRG0–SPRG7), eight pairs of instruction BATs (IBAT0–IBAT7), eight pairs of data BATs (DBAT0–DBAT7), one system version register (SVR), one system memory base address (MBAR), one instruction address breakpoint control (IBCR), one data address breakpoint control (DBCR), a new instruction breakpoint register (IABR2), and two data address breakpoint registers (DABR and DABR2) are integrated into the core.

Supervisor-level SPRs include the following:

- The DSISR defines the cause of data access and alignment interrupts. The cause of a DSI interrupt for a data breakpoint (match with DABR and DABR2) can be determined by the value of the DSISR[DABR] bit (bit 9).
- The data address register (DAR) holds the address of an access after an alignment or DSI interrupt. For example, it contains the address of the breakpoint match condition.
- The decrementer register (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a decrementer interrupt after a programmable delay.
- SDR1 specifies the page table format used in virtual-to-physical address translation for pages. (Note that physical address is referred to as ‘real address’ in the architecture specification.)

- The machine status save/restore register 0 (SRR0) is used for saving the address of the instruction that caused the interrupt, and the address to return to when a Return from Interrupt (**rfi**) instruction is executed.
- The machine status save/restore register 1 (SRR1) is used to save machine status on interrupts and to restore machine status when an **rfi** instruction is executed.
- The SPRG0–SPRG7 registers are provided for operating system use. They reduce the latency that may be incurred in the saving of registers to memory while in a handler. Note that the e300 implements four more SPRGs than the G2 (SPRG0–SPRG3).
- The time base register (TB) is a 64-bit register that maintains the time of day and operates interval timers. It consists of two 32-bit fields: time base upper (TBU) and time base lower (TBL).
- The processor version register (PVR) is a read-only register that identifies the version (model) and revision level of the processor. See Table 1-3 for the version and revision level of the PVR for the e300 processor core.
- Block address translation (BAT) arrays—The PowerPC architecture defines 16 BAT registers. The e300 core includes a total of eight pairs of DBAT and eight pairs of IBAT registers. See Figure 1-4 for a list of the SPR numbers for the BAT arrays.

The following supervisor-level SPRs are implementation-specific (not defined in the PowerPC architecture):

- DMISS and IMISS are read-only registers that are loaded automatically on an instruction or data TLB miss.
- HASH1 and HASH2 contain the physical addresses of the primary and secondary page table entry groups (PTEGs).
- ICMP and DCMP contain a duplicate of the first word in the page table entry (PTE) for which the table search is looking.
- The required physical address (RPA) register is loaded by the core with the second word of the correct PTE during a page table search.
- The system version register (SVR) is available on the e300 core, which identifies the specific version (model) and revision level of the system-on-a-chip (SOC) integration.
- System memory base address (MBAR) is an implementation-specific register available on the e300 core. It supports a temporary storage for the system-level memory map.
- The instruction and data address breakpoint registers (IABR, IABR2, DABR, DABR2) are loaded with an instruction or data address, respectively, that is compared to instruction addresses in the dispatch queue or to the data address in the LSU. When an address match occurs, a breakpoint interrupt is generated.
- One instruction breakpoint control register (IBCR) and one data breakpoint control register (DBCR) are implemented in the e300 core.
- To support critical interrupts, two registers (CSRR0 and CSRR1) are included in the e300 core.
- Eight SPRG registers (SPRG0–SPRG7) are in the e300 core.
- Block address translation (BAT) arrays—The e300 core has eight instruction and eight data BAT registers.

- The hardware implementation (HID0 and HID1) registers provide the means for enabling core checkstops and features and allow software to read the configuration of the PLL configuration signals. The HID2 register enables the true little-endian mode, cache way-locking, and the additional BAT registers.

1.3.2 Instruction Set and Addressing Modes

The following sections describe the PowerPC instruction set and addressing modes in general.

1.3.2.1 PowerPC Instruction Set and Addressing Modes

All PowerPC instructions are encoded as single-word (32-bit) opcodes. Instruction formats are consistent among all instruction types, permitting efficient decoding to occur in parallel with operand accesses. This fixed instruction length and consistent format simplifies instruction pipelining.

The PowerPC instructions are divided into the following categories:

- Integer instructions—These include computational and logical instructions.
  - — Integer arithmetic instructions
  - — Integer compare instructions
  - — Integer logical instructions
  - — Integer rotate and shift instructions
- Floating-point instructions—These include floating-point computational instructions, as well as instructions that affect the FPSCR.
  - — Floating-point arithmetic instructions
  - — Floating-point multiply/add instructions
  - — Floating-point rounding and conversion instructions
  - — Floating-point compare instructions
  - — Floating-point status and control instructions
- Load/store instructions—These include integer and floating-point load and store instructions.
  - — Integer load and store instructions
  - — Integer load and store multiple instructions
  - — Floating-point load and store
  - — Primitives used to construct atomic memory operations (**lwarx** and **stwcx.** instructions)
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow.
  - — Branch and trap instructions
  - — Condition register logical instructions
- Processor control instructions—These instructions are used for synchronizing memory accesses and management of caches, TLBs, and the segment registers.
  - — Move to/from SPR instructions
  - — Move to/from MSR

  - — Move to/from PMR
  - — Synchronize
  - — Instruction synchronize
- Memory control instructions—These instructions provide control of caches, TLBs, and segment registers.
  - — Supervisor-level cache management instructions
  - — Translation lookaside buffer management instructions. Note that there are additional implementation-specific instructions.
  - — User-level cache instructions
  - — Segment register manipulation instructions
- The e300 core implements the following instructions which are defined as optional by the PowerPC architecture:
  - — Floating Select (**fsel**)
  - — Floating Reciprocal Estimate Single-Precision (**fres**)
  - — Floating Reciprocal Square Root Estimate (**frsqrte**)
  - — Store Floating-Point as Integer Word (**stfiwx**)

Note that this grouping of instructions does not indicate the execution unit that executes a particular instruction or group of instructions.

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision (one word) and double-precision (one double word) floating-point operands. The PowerPC architecture uses instructions that are 4 bytes long and word-aligned. It provides for byte, half-word, and word operand loads and stores between memory and a set of 32 GPRs. It also provides for word and double-word operand loads and stores between memory and a set of 32 FPRs.

Computational instructions do not modify memory. To use a memory operand in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written back to the target location with distinct instructions.

The core follows the program flow when it is in the normal execution state. However, the flow of instructions can be interrupted directly by the execution of an instruction or by an asynchronous event. Either kind of interrupt may cause one of several components of the system software to be invoked.

1.3.2.2 Implementation-Specific Instruction Set

The e300 core instruction set is defined as follows:

- The core provides hardware support for all 32-bit PowerPC instructions.
- The core provides two implementation-specific instructions used for software table search operations following TLB misses:
  - — Load Data TLB Entry (**tlbld**)
  - — Load Instruction TLB Entry (**tlbli**)
- The core implements the following instruction which is added to support critical interrupts (also supported on the G2_LE). This is a supervisor-level, context synchronizing instruction.

  - Return from Critical Interrupt (**rfci**)
- The core implements the following instruction which is added to support easy start-up initialization or reloading of the instruction cache.
  - Instruction Cache Block Touch (**icbt**)
- The core provides the following performance monitor instructions:
  - Move to Performance Monitor Register (**mtpmr**)
  - Move from Performance Monitor Register (**mfpmr**)

1.3.3 Cache Implementation

The following sections describe the general cache characteristics as implemented in the PowerPC architecture and the core implementation.

1.3.3.1 PowerPC Cache Characteristics

The PowerPC architecture does not define hardware aspects of cache implementations. The e300 core controls the following memory access modes on a page or block basis:

- Write-back/write-through mode
- Caching-inhibited mode
- Memory coherency

Note that in the core, a cache block is defined as eight words. The VEA defines cache management instructions that provide a means by which the application programmer can affect the cache contents.

1.3.3.2 Implementation-Specific Cache Organization

The e300c1 provide independent, 32-Kbyte, eight-way, set-associative, instruction and data caches. The e300c2 and e300c3 provides 16-Kbyte, four-way set-associative instruction and data caches. The caches are physically addressed, and the data cache can operate in either write-back or write-through mode as specified by the PowerPC architecture.

The data cache is configured as 128 sets of 8 blocks each on the e300c1. The data cache is configured as 128 sets of 4 blocks each on the e300c2 and e300c3. Each block consists of 32 bytes, 2 state bits, and an address tag. The two state bits implement the three-state MEI (modified/exclusive/invalid) protocol. Each block contains eight 32-bit words. Note that the PowerPC architecture defines the term ‘block’ as the cacheable unit. For the core, the block size is equivalent to a cache line. A block diagram of the data cache organization is shown in Figure 1-5.

The instruction cache is configured as 128 sets of 8 blocks each on the e300c1. The instruction cache is configured as 128 sets of 4 blocks each on the e300c2 and e300c3. Each block consists of 32 bytes, an address tag, and a valid bit. The instruction cache may not be written to, except through a block fill operation. In the e300 core, the instruction cache is blocked only until the critical load completes. The e300 core supports instruction fetching from other instruction cache lines following the forwarding of the critical-first-double-word of a cache line load operation. Successive instruction fetches from the cache line being loaded are forwarded, and accesses to other instruction cache lines can proceed during the cache line

load operation. The instruction cache is not snooped, and cache coherency must be maintained by software. A fast hardware invalidation capability is provided to support cache maintenance. The organization of the instruction cache for the e300c1is very similar to the data cache shown in Figure 1-5.

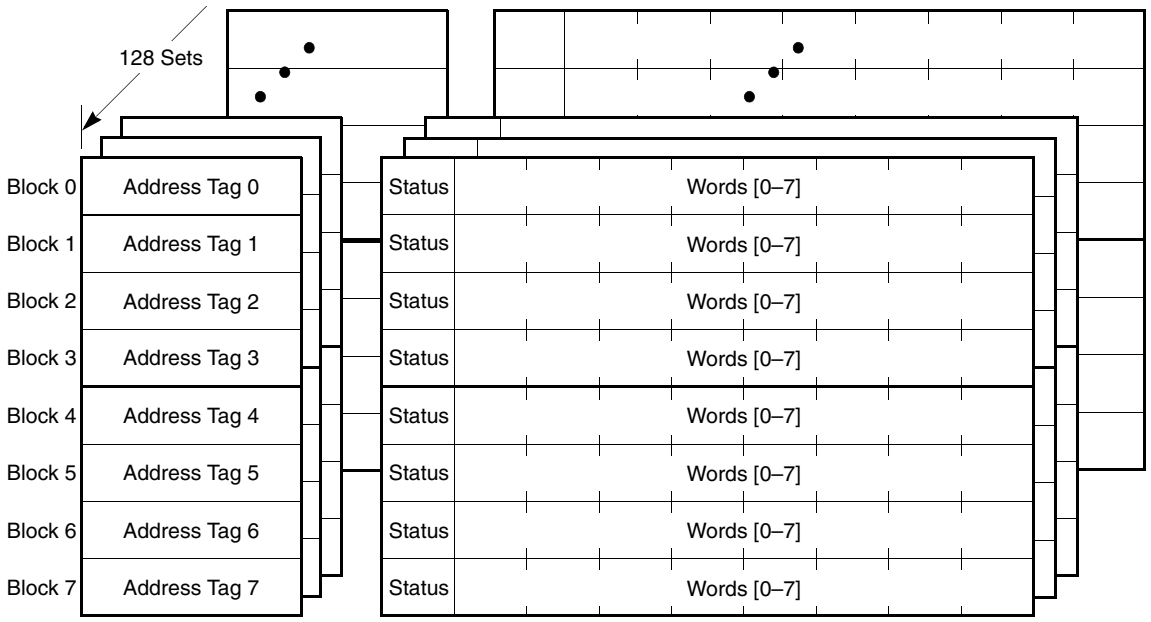


Figure 1-5. e300c1 Data Cache Organization

The e300c2 and e300c3 data cache is configured as 128 sets of four blocks per set. The organization of the data cache is shown in Figure 1-6.

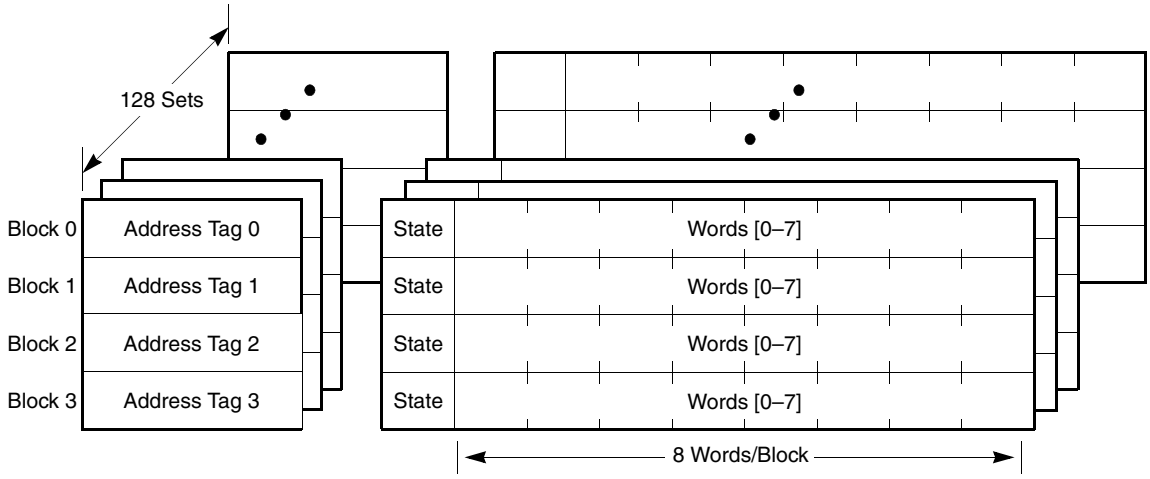


Figure 1-6. e300c2 and e300c3 Data Cache Organization

Each cache block contains eight contiguous words from memory that are loaded from an 8-word boundary (that is, bits A[27–31] of the effective addresses are zero); thus, a cache block never crosses a page boundary. Misaligned accesses across a page boundary can incur a performance penalty.

The e300 core cache blocks are loaded in four beats of 64 bits each on the 64-bit data bus. The burst load is performed as critical-double-word-first. The data cache is blocked to internal accesses until the load

completes; the instruction cache allows sequential fetching during a cache block load. In the core, the critical-double-word is simultaneously written to the cache and forwarded to the requesting unit, thus minimizing stalls due to load delays.

To ensure coherency among caches in a multiprocessor (or multiple caching-device) implementation, the core implements the MEI protocol during normal operation of the data cache. The new data cache MESI extension supports the additional fourth cache coherency shared state for the data cache. To support this feature, the shared signal, *shd*, has been added to the bus interface. The following four states indicate the state of the cache block:

- Modified—The cache block is modified with respect to system memory; that is, data for this address is valid only in the cache and not in system memory.
- Exclusive—This cache block holds valid data that is identical to the data at this address in system memory. No other cache has this data.
- Shared—Only available if HID2[MESISTATE] register bit is set. The address block is valid in the cache and in at least one other cache. This block is always consistent with system memory. That is, the shared state is shared-unmodified; there is no shared-modified state.
- Invalid—This cache block does not hold valid data.

Cache coherency is enforced by on-chip bus snooping logic. Because the e300 core data cache tags are single-ported, a simultaneous load/store and snoop access represents a resource contention. The snoop access is given first access to the tags. The load or store then occurs on the clock following the snoop.

Parity is now integrated into both instruction and data cache memory. A machine check interrupt is now taken upon the detection of an instruction or data cache parity error. Parity is checked whenever valid data is returned from the instruction or data cache for a cache hit or whenever valid data is read out of the cache for a castout or snoop-push operation.

1.3.3.3 Instruction and Data Cache Way-Locking

The e300 core implements instruction and data cache way-locking, which guarantees that certain memory accesses will hit in the cache. This provides deterministic access times for those accesses. See Chapter 4, “Instruction and Data Cache Operation,” for more information.

1.3.4 Interrupt Model

This section describes the PowerPC interrupt model and the e300 core implementation specifically.

1.3.4.1 PowerPC Interrupt Model

The PowerPC interrupt mechanism allows the core to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. The conditions that can cause interrupts are called exceptions. When interrupts occur, information about the state of the core is saved to certain registers and the core begins execution at an address (interrupt vector) predetermined for each interrupt type. Interrupts are processed in supervisor mode.

Some interrupts, such as program interrupts, can be triggered by a broad range of exception conditions. Other interrupts, such as the decrementer interrupt, have only a single exception condition. Exceptions and

the interrupts they cause are described in Chapter 5, “Interrupts and Exceptions.” Although multiple exception conditions can map to a single interrupt vector, a more specific condition may be determined by examining a register associated with the interrupt—for example, the DSISR and the FPSCR. Additionally, some exception conditions can be explicitly enabled or disabled by software.

The PowerPC architecture requires that interrupts be handled in program order; therefore, although a particular implementation may recognize exception conditions out of order, they are presented strictly in order. When an instruction-caused interrupt is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute stage, are required to complete before the interrupt is taken. Any interrupts caused by those instructions are handled first. Likewise, asynchronous, precise interrupts are recognized when they occur, but are not handled until the instruction currently in the completion stage successfully completes execution or generates an interrupt, and the completed store queue is emptied.

Unless a catastrophic condition causes a system reset or machine check interrupt, only one interrupt is handled at a time. If, for example, a single instruction encounters multiple interrupt conditions, those conditions are handled sequentially. After the interrupt handler completes, the instruction execution continues until the next interrupt condition is encountered. However, in many cases there is no attempt to re-execute the instruction. This method of recognizing and handling interrupts sequentially guarantees that interrupts are recoverable.

To prevent the program state from being lost due to a system reset, a machine check interrupt, or an instruction-caused interrupt in the interrupt handler, interrupt handlers should save the information stored in SRR0 and SRR1 early and before enabling external interrupts.

The PowerPC architecture supports four types of interrupts:

- Synchronous, precise—These are caused by instructions. All instruction-caused interrupts are handled precisely; that is, the machine state at the time the interrupt occurs is known and can be completely restored. This means that (excluding the trap and system call interrupts) the address of the faulting instruction is provided to the interrupt handler and neither the faulting instruction nor subsequent instructions in the code stream will complete execution before the interrupt is taken. Once the interrupt is processed, execution resumes at the address of the faulting instruction (or at an alternate address provided by the interrupt handler). When an interrupt is taken due to a trap or system call instruction, execution resumes at an address provided by the handler.
- Synchronous, imprecise—The PowerPC architecture defines two imprecise floating-point exception modes: recoverable and nonrecoverable. Even though the core provides a means to enable the imprecise modes, it implements these modes identically to the precise mode (that is, all enabled floating-point exceptions are always precise on the core).
- Asynchronous, maskable—The external, system management interrupt (SMI) and decrementer interrupts are maskable, asynchronous interrupts. When these interrupts occur, their handling is postponed until the next instruction and any of its associated interrupts complete execution. If there are no instructions in the execution units, the interrupt is taken immediately upon determination of the correct restart address (for loading SRR0).
- Asynchronous, nonmaskable—The system reset and the machine check interrupt are nonmaskable, asynchronous interrupts. They may not be recoverable, or they may provide a limited degree of recoverability. All interrupts report recoverability through MSR[RI].

1.3.4.2 Implementation-Specific Interrupt Model

As specified by the PowerPC architecture, all interrupts can be described as either precise or imprecise and either synchronous or asynchronous. Asynchronous interrupts (some of which are maskable) are caused by events external to the processor's execution; synchronous interrupts, which are all handled precisely by the e300 core, are caused by instructions. A system management interrupt is an implementation-specific interrupt. The interrupt classes are shown in Table 1-1. The interrupts are listed in Table 5-3 in order of highest to lowest priority.

Table 1-1. Interrupt Classifications

Synchronous/Asynchronous	Precise/Imprecise	Interrupt Type
Asynchronous, nonmaskable	Imprecise	Machine check System reset
Asynchronous, maskable	Precise	External interrupt Decrementer System management interrupt Critical interrupt
Synchronous	Precise	Instruction-caused interrupts

Although interrupts have other characteristics, such as whether they are maskable, the distinctions shown in Table 1-1 define categories of interrupts that the core handles uniquely. Note that Table 1-1 includes no synchronous, imprecise instructions. While the PowerPC architecture supports imprecise handling of floating-point exceptions, the core implements floating-point exception modes as precise.

The e300 core interrupts and exception conditions that cause them are listed in Table 1-2.

Table 1-2. Exceptions and Interrupts

Interrupt Type	Vector Offset (hex)	Exception Conditions
Reserved	00000	—
System reset	00100	Caused by the assertion of either $\overline{sreset}$ or $\overline{hreset}$.
Machine check	00200	Caused by the assertion of the $\overline{tea}$ signal during a data bus transaction, assertion of $\overline{mcp}$, an address or data parity error, or an instruction or data cache parity error. Note that the e300 has SRR1 register values that are different from the G2/G2_LE cores' when a machine check occurs. See Table 5-14 for more information.
DSI	00300	Determined by the bit settings in the DSISR, listed as follows: 1 Set if the translation of an attempted access is not found in the primary hash table entry group (HTEG), or in the rehashed secondary HTEG, or in the range of a DBAT register; otherwise cleared 4 Set if a memory access is not permitted by the page or DBAT protection mechanism; otherwise cleared 6 Set for a store operation and cleared for a load operation 9 Set if a data address breakpoint interrupt occurs when the data [0–28] in the DABR or DABR2 matches the next data access (load or store instruction) to complete in the completion unit. The different breakpoints are enabled as follows: • Write breakpoints enabled when DABR[30] is set • Read breakpoints enabled when DABR[31] is set

Table 1-2. Exceptions and Interrupts (continued)

Interrupt Type	Vector Offset (hex)	Exception Conditions
ISI	00400	Caused when an instruction fetch cannot be performed for any of the following reasons: • The effective (logical) address cannot be translated. That is, there is a page fault for this portion of the translation, so an ISI interrupt must be taken to load the PTE (and possibly the page) into memory. • The fetch access violates memory protection (indicated by SRR1[4] set). If the key bits (Ks and Kp) in the segment register and the PP bits in the PTE are set to prohibit read access, instructions cannot be fetched from this location.
External interrupt	00500	Caused when MSR[EE] = 1 and the $\overline{int}$ signal is asserted.
Alignment	00600	Caused when the core cannot perform a memory access for any of the reasons described below: • The operand of a floating-point load or store instruction is not word-aligned. • The operands of **lmw**, **stmw**, **lwarx**, and **stwcx.** instructions are not aligned. • The instruction is **lswi**, **lswx**, **stswi**, **stswx**, and the core is in little-endian mode. Note that PowerPC little-endian mode is not supported on the e300 core. • The operand of **dcbz** is in memory that is write-through-required or caching-inhibited.
Program	00700	Caused by one of the following exception conditions, which correspond to bit settings in SRR1 and arise during execution of an instruction. Floating-point enabled exception—A floating-point enabled exception condition is generated when the following condition is met: (MSR[FE0] | MSR[FE1]) and FPSCR[FEX] is 1. • FPSCR[FEX] is set by the execution of a floating-point instruction that causes an enabled exception or by the execution of one of the Move to FPSCR instructions that results in both an exception condition bit and its corresponding enable bit being set in the FPSCR. • Illegal instruction—An illegal instruction program interrupt is generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields (including PowerPC instructions not implemented in the core), or when execution of an optional instruction not provided in the core is attempted (these do not include those optional instructions that are treated as no-ops). • Privileged instruction—A privileged instruction program interrupt is generated when the execution of a privileged instruction is attempted and the MSR register user privilege bit, MSR[PR], is set. In the e300 core, this interrupt is generated for **mtspr** or **mfspr** with an invalid SPR field if SPR[0] = 1 and MSR[PR] = 1. This may not be true for all cores that implement the PowerPC architecture. • Trap—A trap type program interrupt is generated when any of the conditions specified in a trap instruction are met.
Floating-point unavailable	00800	Caused by an attempt to execute a floating-point instruction (including floating-point load, store, and move instructions) when the floating-point available bit (MSR[FP]) is cleared. In the e300c2 core, any attempt to execute a floating-point instruction results in a floating-point unavailable exception.
Decrementer	00900	Occurs when DEC[31] changes from 0 to 1. This interrupt is enabled with MSR[EE].
Critical interrupt	00A00	Taken when $\overline{cint}$ is asserted and MSR[CE] = 1.
Reserved	00B00–00BFF	—
System call	00C00	Occurs when a System Call (**sc**) instruction is executed.
Trace	00D00	Taken when MSR[SE] =1 or when the currently completing instruction is a branch and MSR[BE] =1.

Table 1-2. Exceptions and Interrupts (continued)

Interrupt Type	Vector Offset (hex)	Exception Conditions
Reserved	00E00	The e300 core does not generate an interrupt to this vector. Other devices may use this vector for floating-point assist interrupts.
Performance monitor	00F00	Caused when *pm_event_in* is asserted.
Instruction translation miss	01000	Caused when the effective address for an instruction fetch cannot be translated by the ITLB.
Data load translation miss	01100	Caused when the effective address for a data load operation cannot be translated by the DTLB.
Data store translation miss	01200	Caused when the effective address for a data store operation cannot be translated by the DTLB, or when a DTLB hit occurs and the change bit in the PTE must be set due to a data store operation.
Instruction address breakpoint	01300	Occurs when the address (bits 0–29) in the IABR matches the next instruction to complete in the completion unit, and IABR[30] is set. Note that the e300 core also implements IABR2, which functions identically to IABR.
System management interrupt	01400	Caused when MSR[EE] = 1 and the $\overline{smi}$ input signal is asserted.
Reserved	01500–02FFF	—

1.3.5 Memory Management

The following sections describe the memory management features of the PowerPC architecture and the e300 core implementation, respectively.

1.3.5.1 PowerPC Memory Management

The primary functions of the MMU are to translate logical (effective) addresses to physical addresses for memory accesses and to provide access protection on blocks and pages of memory.

The core generates two types of accesses that require address translation: instruction accesses and data accesses to memory generated by load and store instructions.

The PowerPC MMU and interrupt model support demand-paged virtual memory. Virtual memory management permits execution of programs larger than the size of physical memory; demand-paged implies that individual pages are loaded into physical memory from system memory only when they are first accessed by an executing program.

The hashed page table is a variable-sized data structure that defines the mapping between virtual page numbers and physical page numbers. The page table size is a power of two, and its starting address is a multiple of its size.

The page table contains a number of page-table entry groups (PTEGs). A PTEG contains eight page-table entries (PTEs) of 8 bytes each; therefore, each PTEG is 64 bytes long. PTEG addresses are entry points for table search operations.

Address translations are enabled by setting bits in the MSR—MSR[IR] enables instruction address translations, and MSR[DR] enables data address translations.

1.3.5.2 Implementation-Specific Memory Management

The instruction and data memory management units in the e300 core provide 4 Gbytes of logical address space accessible to supervisor and user programs with a 4-Kbyte page size and 256-Mbyte segment size. Block sizes range from 128 Kbytes to 256 Mbytes and are software selectable. In addition, the core uses an interim 52-bit virtual address and hashed page tables for generating 32-bit physical addresses. The MMUs in the e300 core rely on the interrupt processing mechanism for the implementation of the paged virtual memory environment and for enforcing protection of designated memory areas.

Instruction and data TLBs provide address translation in parallel with the on-chip cache access, incurring no additional time penalty in the event of a TLB hit. A TLB is a cache of the most recently used page table entries. Software is responsible for maintaining the consistency of the TLB with memory. The core TLBs are 64-entry, two-way, set-associative caches that contain instruction and data address translations. The core provides hardware assist for software table search operations through the hashed page table on TLB misses. Supervisor software can invalidate TLB entries selectively.

For instructions and data that correspond to block address translation, the e300 core provides independent eight-entry BAT arrays. These entries define blocks that can vary from 128 Kbytes to 256 Mbytes. The BAT arrays are maintained by system software. HID2[HBE] is added to the e300 for enabling or disabling the four additional pairs of BAT registers. However, regardless of the setting of HID2[HBE], these BATs are accessible by **mfspr** and **mtspr**.

As specified by the PowerPC architecture, the hashed page table is a variable-sized data structure that defines the mapping between virtual page numbers and physical page numbers. The page table size is a power of two, and its starting address is a multiple of its size.

Also as specified by the PowerPC architecture, the page table contains a number of PTEGs. A PTEG contains 8 PTEs of 8 bytes each; therefore, each PTEG is 64 bytes long. PTEG addresses are entry points for table search operations.

1.3.6 Instruction Timing

The e300 core is a pipelined superscalar processor core. Because instruction processing is reduced into a series of stages, an instruction does not require all of the resources of an execution unit at the same time. For example, after an instruction completes the decode stage, it can pass on to the next stage, while the subsequent instruction can advance into the decode stage. This improves the throughput of the instruction flow. For example, it may take three cycles for a single floating-point instruction to execute, but if there are no stalls in the floating-point pipeline, a series of floating-point instructions can have a throughput of one instruction per cycle.

The core instruction pipeline has four major pipeline stages, described as follows:

- The fetch pipeline stage primarily involves retrieving instructions from the memory system and determining the location of the next instruction fetch. Additionally, if possible, the BPU decodes branches during the fetch stage and folds out branch instructions before the dispatch stage.

- The dispatch pipeline stage is responsible for decoding the instructions supplied by the instruction fetch stage and determining which of the instructions are eligible to be dispatched in the current cycle. In addition, the source operands of the instructions are read from the appropriate register file and dispatched with the instruction to the execute pipeline stage. At the end of the dispatch pipeline stage, the dispatched instructions and their operands are latched by the appropriate execution unit.
- In the execute pipeline stage, each execution unit with an instruction executes the selected instruction (perhaps over multiple cycles), writes the instruction's result into the appropriate rename register, and notifies the completion stage when the execution has finished. In the case of an internal interrupt, the execution unit reports the interrupt to the completion/write-back pipeline stage and discontinues instruction execution until the interrupt is handled. The interrupt is not signaled until that instruction is the next to be completed. Execution of most floating-point instructions is pipelined within the FPU, allowing up to three instructions to execute in the FPU concurrently. The FPU pipeline stages are multiply, add, and round-convert. The LSU has two pipeline stages: the first stage, for effective address calculation and MMU translation, and the second, for accessing data in the cache.
- The complete/write-back pipeline stage maintains the correct architectural machine state and transfers the contents of the rename registers to the GPRs and FPRs as instructions are retired. If the completion logic detects an instruction causing an interrupt, all subsequent instructions are canceled, their execution results in rename registers are discarded, and instructions are fetched from the correct instruction stream.

A superscalar processor core issues multiple, independent instructions into multiple pipelines, allowing instructions to execute in parallel. The e300c1 core has independent execution units for: integer instructions, floating-point instructions, branch instructions, load/store instructions, and system register instructions.The e300c2 does not include a floating-point unit. The e300c2 and e300c3 provide two IUs, which improves the throughput of integer instructions. The e300c2 and e300c3 provide two integer units for greater integer instruction throughput along with enhanced multipliers in each IU that reduce the multiply instruction latency to a maximum of two cycles. The IU and the FPU each have dedicated register files for maintaining operands (GPRs and FPRs, respectively), allowing integer and floating-point calculations to occur simultaneously without interference. The e300c2 does not include floating-point registers.

The core provides support for single-cycle store, and it provides an adder/comparator in the system register unit that allows the dispatch and execution of multiple integer add and compare instructions on each cycle. Refer to Chapter 7, “Instruction Timing,” for more information.

Because the PowerPC architecture can be applied to such a wide variety of implementations, instruction timing among processor cores varies accordingly.

1.3.7 Core Interface

The core interface is specific for each processor core implementation.

The e300 core provides a versatile core interface that allows for a wide range of implementations. The interface includes a 32-bit address bus, a 64-bit data bus, and 56 control and information signals (see Figure 1-7). The core interface allows for address-only transactions, as well as address and data transactions. The core control and information signals include the address arbitration, address start,

address transfer, transfer attribute, address termination, data arbitration, data transfer, data termination, and core state signals. Test and control signals provide diagnostics for selected internal circuits.

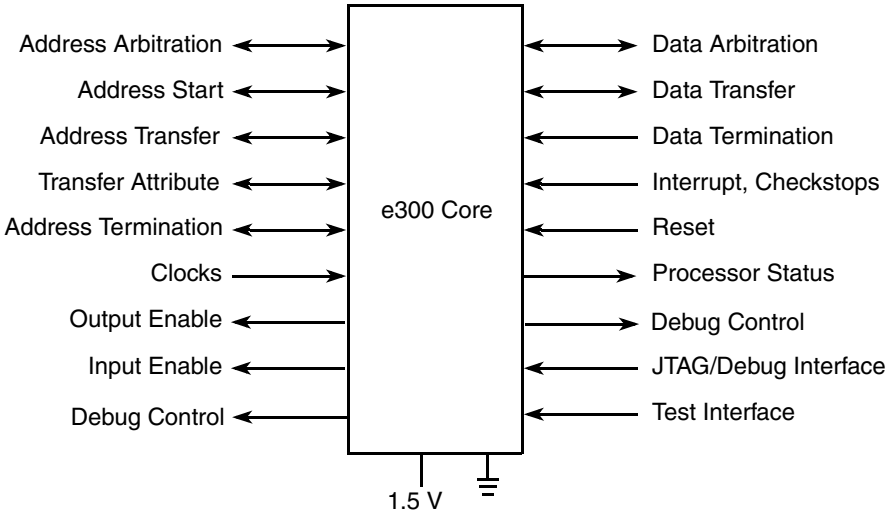


Figure 1-7. Core Interface

The core interface supports bus pipelining, allowing the address tenure of one transaction to overlap the data tenure of another. The extent of the pipelining depends on external arbitration and control circuitry. Similarly, the core supports split-bus transactions for systems with multiple potential bus masters—one device can have mastership of the address bus while another has mastership of the data bus. Allowing multiple bus transactions to occur simultaneously increases the available bus bandwidth for other activity and, as a result, improves performance.

The core clocking structure allows the bus to operate at integer multiples of the core cycle time.

The following sections describe the core bus support for memory operations. Note that some signals perform different functions depending on the addressing protocol used.

The e300c4 can optionally support the core complex bus (CCB). See Chapter 8, “Core Interface Operation,” for more information on the CCB.

1.3.7.1 Memory Accesses

The e300 core CSB is a 64-bit data bus.

With a 64-bit CSB, memory accesses allow transfer sizes of 8, 16, 24, 32, 40, 48, 56, or 64 bits in one bus clock cycle. Data transfers occur in either single-beat transactions or four-beat burst transactions. Single-beat transactions are caused by noncached accesses that access memory directly (that is, reads and writes when caching is disabled, caching-inhibited accesses, and stores in write-through mode). Four-beat burst transactions, which always transfer an entire cache block (32 bytes), are initiated when a line is read from or written to memory.

1.3.7.2 Signals

The e300 core signals are grouped as follows:

- Interrupts/Resets—These signals include the external interrupt signal (*int*), critical interrupt signal (*cint*), checkstop signals, performance monitor signal (*pm_event_in*), and both soft reset and hard reset signals. They are used to interrupt and, under various conditions, to reset the core.
- JTAG/debug interface signals—The JTAG (based on **the IEEE** 1149.1 standard) interface and debug unit provides a serial interface to the system for performing monitoring and boundary tests. Two additional signals are added to the e300 core to allow observation of the internal clock state of the core (*stopped*) and to allow the external input to force the core into a halted state (*ext_halt*).
- Core status and control—These signals include the memory reservation signal, machine quiesce control signals, time base enable signal, and the *tlbisync* signal.
- Clock control—These signals provide for system clock input and frequency control.
- Test interface signals—Signals like address matching, combinational matching, and watchpoint are used in the core for production testing.
- Transfer attribute signals—These signals provide information about the type of transfer, such as the transfer size and whether the transaction is bursted, write-through, or cache-inhibited.

NOTE

A bar over a signal name indicates that the signal is active low—for example, *artry_in* (address retry) and *ts_in* (transfer start). Active-low signals are referred to as asserted (active) when they are low and negated when they are high. Signals that are not active low, such as *ap_in[0:3]* (address bus parity signals) and *tt_in[0:4]* (transfer type signals) are referred to as asserted when they are high and negated when they are low.

1.3.8 Debug Features

Some new debug features are specific to the e300 core. Accesses to the debug facilities are available only in supervisor mode by using the **mtspr** and **mfspr** instructions. The e300 provides the following additional feature in the JTAG/debug interface: Inclusion of breakpoint status and control pins: *stopped* and *ext_halt.*

1.3.8.1 Breakpoint Signaling

The breakpoint signaling provided on the e300 core allows observability of breakpoint matches external to the core. The *iabr, iabr2, dabr,* and *dabr2* breakpoint signals are asserted for at least one bus clock cycle when the respective breakpoint occurs. The status of the run state of the e300 core is indicated by the *stopped* pin. An asynchronous external breakpoint can be asserted to the e300 core using the *ext_halt* pin:

- When DBCR and IBCR are configured for an OR combinational signal type, the breakpoint signals *iabr, iabr2* and *dabr, dabr2* reflect their respective breakpoints.
- When the DBCR and IBCR are configured for AND combinational signal type, only the *iabr2* and *dabr2* breakpoint signals are asserted after the AND condition is met (that is, both instruction breakpoints occurred or both data breakpoints occurred).

- When the core_stopped pin is asserted, the e300 core has entered a stopped state and all internal clocking has stopped, indicating that a hardware debug event has occurred.
- The *ext_halt* input pin can be used to force the core into halted state. The halted state may be a hardstop, conditional upon the HARDSTOP condition being set through the JTAG/debug interface

The breakpoint signaling conditions are described in Chapter 10, "Debug Features."

1.4 Differences Between Cores

The e300 core has similar functionality to the G2_LE core. Table 1-3 describes the differences between the G2_LE and the e300.

Table 1-3. Differences Between e300 and G2_LE Cores

e300 Core	G2_LE Core	Impact
New HID0 bits	—	The e300 core has a new HID0 bit defined to enable cache parity error reporting (ECPE).
New HID1 bits	—	The e300 core has new HID1 bits defined to extend the number of PLL configuration signals to seven (PC5, PC6).
New HID2 bits	—	The e300 core has new HID2 bits defined to support instruction fetch bursting (IFEB), MESI coherency protocol (MESI), instruction fetch cancels (IFEC), data cache queue sharing (EBQS), pipelining extension (EBPX), additional cache way locking (IWLCK and DWLCK), and instruction cache way protection (ICWP).
New PVR register value	—	The processor version register values differ. See Table 1-4 for e300 PVR values.
New IBCR and DBCR bits	—	The e300 core has new IBCR[IABRSTAT, IABR2STAT] and DBCR[DABR1STAT, DABR2STAT] fields to provide instruction and data address breakpoint status.
—	16-Kbyte, four-way, set-associative, instruction and data caches	Some e300 cores may have different cache sizes than the G2_LE.
L1 cache parity	—	The e300 core supports parity for both instruction and data caches; the G2_LE does not support cache parity.
MEI or MESI coherency protocols	MEI protocol only	The e300 supports two coherency protocols: MEI and MESI; the G2_LE only supports the MEI protocol.
Instruction cancel extension	—	The e300 instruction cancel mechanism improves utilization of instruction cache by supporting 'hits-under-cancels' and 'misses-under-cancels'; the G2_LE requires the cancel to complete before new instruction fetches can begin.

Table 1-3. Differences Between e300 and G2_LE Cores (continued)

e300 Core	G2_LE Core	Impact
Instruction fetch bursts to caching-inhibited space	Single-beat instruction fetches to caching-inhibited space	The e300's instruction fetch burst extension allows all caching-inhibited instruction fetches to be performed on the bus as burst transactions, even though the instructions are not cached. This improves performance for instruction space that is caching-inhibited, because up to eight instructions are returned with one bus operation. The G2_LE core must use single-beat instruction fetches for caching-inhibited space, returning only two instructions per bus operation.
Instruction cache way protection	—	The e300 core can protect locked ways in the instruction cache from invalidation; the G2_LE does not support instruction cache way protection.
Data cache queue sharing	—	The e300 has a new data cache queue sharing extension that allows the two burst-write queues in the bus unit to be used interchangeably for cache replacements and snoop pushes. Thus, the data cache can support two outstanding cache replacements or two outstanding snoop push operations on the bus at any given time.
icbt instruction	—	The e300 supports a new instruction cache block touch instruction that facilitates preloading the instruction cache before locking; the G2_LE core requires speculatively fetching instructions before locking the instruction cache.
1-1/2-level bus pipelining	1-level bus pipelining	For the e300, a new transaction can complete an address tenure when the previous transaction has been granted the data bus; for the G2_LE, a new transaction must wait until the previous data tenure has completed before completing its address tenure.
PowerPC little-endian not supported	PowerPC little-endian supported	PowerPC little-endian will not be supported in the e300 core, although true little-endian will be fully supported.
Data retry mode removed	Data retry mode available	$\overline{drtry}$ and *drtrymode* will no longer be supported on the e300 and future versions.
External control instructions removed	External control instructions available	The **eciwx** and **ecowx** instruction pair will not be supported on the e300 core. These are optional instructions in the PowerPC architecture.
Reduced pin mode removed	Reduced pin mode available	Reduced pinout mode and the signal *redpinmode* will not be supported in the e300 core.

1.5 Differences Between e300 Cores

Table 1-4 describes the differences between the e300 cores.

Table 1-4. Differences Between e300 Cores

	Cache Sizes	Floating-Point	Integer Units	Enhanced Multipliers	Performance Monitor	PVR
e300c1	32Kbyte, 8-way, set-associative instruction and data caches	Supported	One	Not included	Not included	0x8083
e300c2	16 Kbyte, 4-way, set-associative instruction and data caches	Not supported	Two	Included	Not included	0x8084
e300c3	16 Kbyte, 4-way, set-associative instruction and data caches	Supported	Two	Included	Included	0x8085
Impact	Some cores implement smaller L1 instruction and data caches.	The e300c2 does not implement hardware support for floating-point operations.	Two integer execution units provide higher throughput of integer operations.	The enhanced multipliers are faster and provide a maximum two-cycle latency for multiply instructions.	A performance monitor provides the ability to count predefined events and processor clocks associated with particular operations, such as cache misses, mispredicted branches, or the number of cycles an execution unit stalls.	The core version number in the processor version register (PVR) is listed in this table, and the revision level for each core starts at 0x0010 and changes for each revision of the core.

Chapter 2
Register Model

This chapter describes the PowerPC register model and its specific implementation on the e300 core. First, it outlines the register organization as defined by the three levels of the PowerPC architecture: user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA). Secondly, this chapter describes the core implementation-specific registers. Full descriptions of the basic register set defined by the PowerPC architecture are provided in Chapter 2, "Register Set," in the *Programming Environments Manual*.

The PowerPC architecture defines register-to-register operations for all computational instructions. Source data for these instructions is accessed from the on-chip registers or is provided as an immediate value embedded in the opcode. The three-register instruction format allows specification of a target register distinct from the two source registers, thus preserving the original data for use by other instructions and reducing the number of instructions required for certain operations. Data is transferred between memory and registers with explicit load and store instructions only.

Note that there may be registers common to other processors of this family that are not implemented in the e300 core. When the core detects special-purpose register (SPR) encodings other than those defined in this document, it either takes an interrupt or it treats the instruction as a no-op. Conversely, some SPRs in the e300 core may not be implemented in other processors or may not be implemented in the same way.

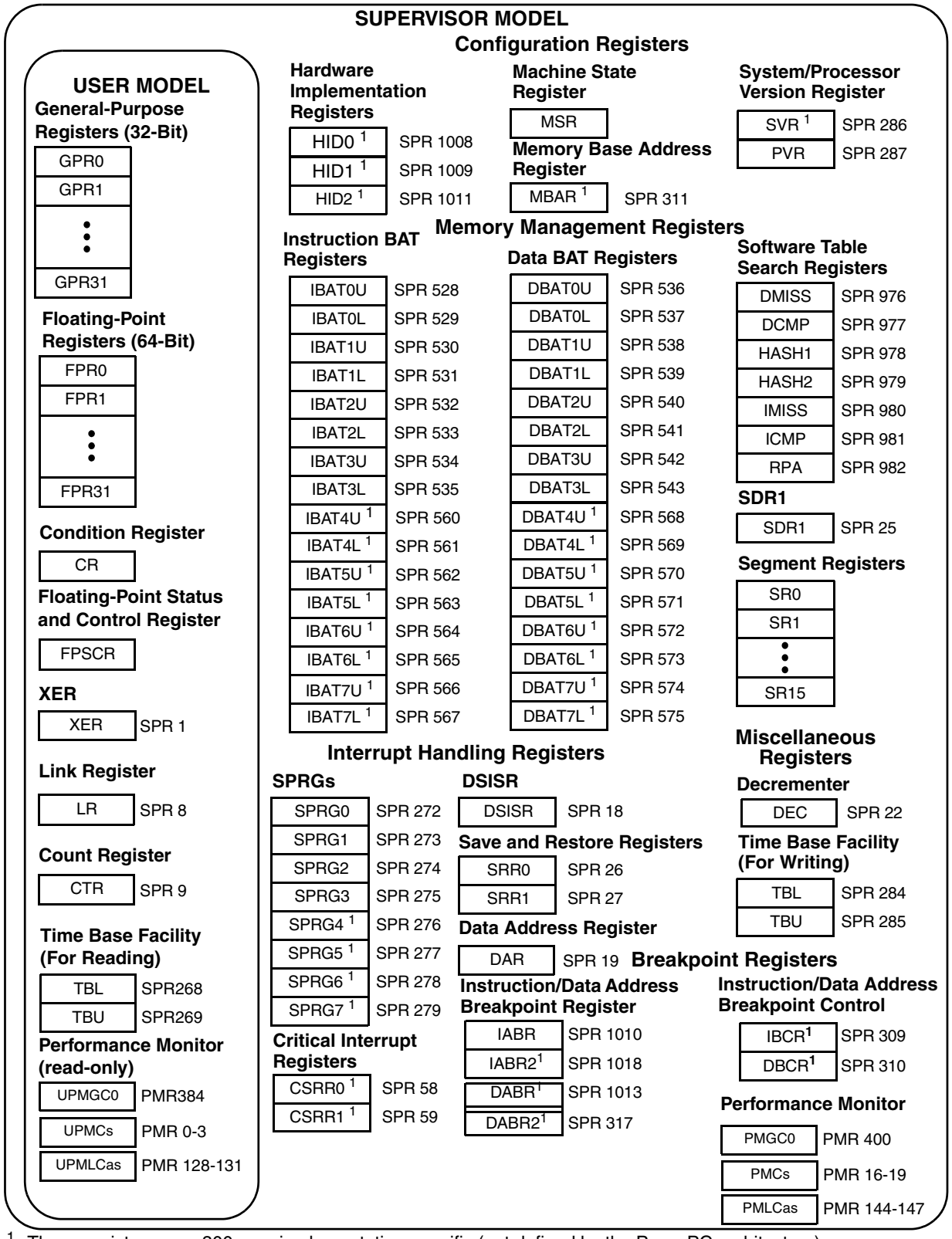
2.1 PowerPC Register Set

The UISA registers, shown in Figure 2-1, can be accessed by either user- or supervisor-level instructions (the architecture specification refers to user- and supervisor-level as problem state and privileged state, respectively). The general-purpose registers (GPRs) and floating-point registers (FPRs) are accessed through instruction operands. Floating-point registers are not supported by the e300c2 core. Access to registers can be explicit (that is, through the use of specific instructions for that purpose, such as the **mtspr** and **mfspr** instructions) or implicit as part of the execution (or side effect) of an instruction. Some registers are accessed both explicitly and implicitly.

Figure 2-1 describes the registers in the e300 core. Note that the implementation-specific registers for the e300 core are shown in Figure 2-1.

The number to the right of the register name indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the XER is SPR1).

For more information on the PowerPC register set, refer to Chapter 2, "Register Set," in the *Programming Environments Manual*.



1 These registers are e300 core implementation-specific (not defined by the PowerPC architecture).

Figure 2-1. e300 Programming Model—Registers

The e300 core user-level registers are described as follows:

- User-level registers (UISA)—The user-level registers can be accessed by all software with either user or supervisor privileges. The user-level register set includes the following:
  - — General-purpose registers (GPRs). The GPR file consists of thirty-two 32-bit GPRs designated as GPR0–GPR31. This register file serves as the data source or destination for all integer instructions and provides data for generating addresses.
  - — Floating-point registers (FPRs). The FPR file consists of thirty-two 64-bit FPRs designated as FPR0–FPR31, which serves as the data source or destination for all floating-point instructions. These registers can contain data objects of either single- or double-precision floating-point format.

    Before the **stfd** instruction is used to store the contents of an FPR to memory, the FPR must have been initialized after reset (explicitly loaded with any value) by using a floating-point load instruction.

    Implementation Note—The e300c2 core does not support any floating point registers or instructions.
  - — Condition register (CR). The CR consists of 4-bit fields, CR0–CR7, that reflect the results of certain arithmetic operations and provides a mechanism for testing and branching.
  - — Floating-point status and control register (FPSCR). The FPSCR contains all floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard. Figure 2-2 shows the bit fields of the FPSCR. The FPSCR is not supported on the e300c2 core. For more detailed information on the FPSCR see the *Programming Environments Manual*.

Access: User read/write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	FX	FEX	VX	OX	UX	ZX	XX	VXSNAN	VXISI	VXIDI	VXZDZ	VXIMZ	VXVC	FR	FI	FPRF		—	VXSOFT	VXSQRT	VXCVI	VE	OE	UE	ZE	XE	NI	RN	
Reset	All zeros																												

Figure 2-2. Floating-Point Status and Control Register (FPSCR)

FPSCR bits are described in Table 2-1

Table 2-1. FPSCR Bit Settings

Bits	Name	Description
0	FX	Floating-point exception summary. Every floating-point instruction, except **mtfsfi** and **mtfsf**, implicitly sets FX if that instruction causes any FPSCR floating-point exception bit to transition from 0 to 1. The **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** instructions can alter FX explicitly. This is a sticky bit.
1	FEX	Floating-point enabled exception summary. Signals the occurrence of any enabled exception conditions. It is the logical OR of all the floating-point exception bits masked by their respective enable bits (FEX = (VX & VE) ^ (OX & OE) ^ (UX & UE) ^ (ZX & ZE) ^ (XX & XE)). The **mcrfs**, **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot alter FPSCR[FEX] explicitly. This is not a sticky bit.
2	VX	Floating-point invalid operation exception summary. This bit signals the occurrence of any invalid operation exception. It is the logical OR of all of the invalid operation exception bits. The **mcrfs**, **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot alter FPSCR[VX] explicitly. This is not a sticky bit.
3	OX	Floating-point overflow exception. This is a sticky bit.
4	UX	Floating-point underflow exception. This is a sticky bit.
5	ZX	Floating-point zero divide exception. This is a sticky bit.
6	XX	Floating-point inexact exception. This is a sticky bit. XX is the sticky version of FPSCR[FI]. A given instruction sets XX as follows: • If the instruction affects FPSCR[FI], the new value of FPSCR[XX] is obtained by logically ORing the old value of FPSCR[XX] with the new value of FPSCR[FI]. • If the instruction does not affect FPSCR[FI], the value of FPSCR[XX] is unchanged.
7	VXSNAN	Floating-point invalid operation exception for SNaN. This is a sticky bit.
8	VXISI	Floating-point invalid operation exception for ∞ – ∞. This is a sticky bit.”
9	VXIDI	Floating-point invalid operation exception for ∞ ÷ ∞. This is a sticky bit.
10	VXZDZ	Floating-point invalid operation exception for 0 ÷ 0. This is a sticky bit.
11	VXIMZ	Floating-point invalid operation exception for ∞ * 0. This is a sticky bit.
12	VXVC	Floating-point invalid operation exception for invalid compare. This is a sticky bit.
13	FR	Floating-point fraction rounded. The last arithmetic, rounding, or conversion instruction incremented the fraction. This bit is not sticky.
14	FI	Floating-point fraction inexact. The last arithmetic, rounding, or conversion instruction either produced an inexact result during rounding or caused a disabled overflow exception. This is not a sticky bit. For more information regarding the relationship between FPSCR[FI] and FPSCR[XX], see the description of the FPSCR[XX] bit.

Table 2-1. FPSCR Bit Settings (continued)

Bits	Name	Description
15–19	FPRF	Floating-point result flags. For arithmetic, rounding, and conversion instructions, FPRF is based on the result placed into the target register, except that if any portion of the result is undefined, the value placed here is undefined. 15 Floating-point result class descriptor (C). Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits to indicate the class of the result. Bits 16–19 comprise the floating-point condition code (FPCC). Floating-point compare instructions always set one of the FPCC bits to one and the other three FPCC bits to zero. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit to indicate the class of the result. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero. 16 Floating-point less than or negative (FL or <) 17 Floating-point greater than or positive (FG or >) 18 Floating-point equal or zero (FE or =) 19 Floating-point unordered or NaN (FU or ?) Note that these are not sticky bits.
20	—	Reserved, should be cleared.
21	VXSOFT	Floating-point invalid operation exception for software request. This is a sticky bit. This bit can be altered only by the **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, or **mtfsb1** instructions.
22	VXSQRT	Floating-point invalid operation exception for invalid square root. This is a sticky bit.
23	VXCVI	Floating-point invalid operation exception for invalid integer convert. This is a sticky bit.
24	VE	Floating-point invalid operation exception enable.
25	OE	IEEE floating-point overflow exception enable.
26	UE	IEEE floating-point underflow exception enable.
27	ZE	IEEE floating-point zero divide exception enable.
28	XE	Floating-point inexact exception enable.
29	NI	Floating-point non-IEEE mode. If this bit is set, results need not conform with IEEE standards and the other FPSCR bits may have meanings other than those described here. If the bit is set and if all *i*mplementation-specific requirements are met and if an IEEE-conforming result of a floating-point operation would be a denormalized number, the result produced is zero (retaining the sign of the denormalized number). Any other effects associated with setting this bit are described in the user's manual for the implementation. Effects of the setting of this bit are implementation-dependent.
30–31	RN	Floating-point rounding control. 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward –infinity

The remaining user-level registers are SPRs. Note that the PowerPC architecture provides a separate mechanism for accessing SPRs (the **mtspr** and **mfspr** instructions). These instructions are commonly used to explicitly access certain registers, while other SPRs may be accessed as the side effect of executing other instructions.

— XER register (XER). The 32-bit XER indicates overflow and carries for integer operations. It is set implicitly by many instructions.

— Link register (LR). The 32-bit LR provides the branch target address for the Branch Conditional to Link Register (**bclr***x*) instruction and can optionally be used to hold the logical address (referred to as the effective address in the architecture specification) of the instruction that follows a branch and link instruction, typically used for linking to subroutines.

— Count register (CTR). The 32-bit CTR can be used to hold a loop count that can be decremented during execution of appropriately coded branch instructions. It can also provide the branch target address for the Branch Conditional to Count Register (**bcctr***x*) instruction.

- User-level registers (VEA)—The VEA introduces the time base facility (TB) for reading. The TB is a 64-bit register pair whose contents are incremented once every four core input clock cycles. The TB consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). Note that the time base registers are read-only in user state.

The core supervisor-level registers are described as follows:

- **Supervisor-level registers (OEA)—The** OEA defines the registers an operating system uses for memory management, configuration, and interrupt handling. The PowerPC architecture defines the following supervisor-level registers:

  — Configuration registers

  – Processor version register (PVR). This read-only register identifies the version (model) and revision level of this processor core. The contents of the PVR can be copied to a GPR by the **mfspr** instruction. Read access to the PVR is supervisor-level only; write access is not provided. The PVR consists of the fields as described in Table 2-2. Architecturally, the PVR consists of two 16-bit fields as described in Table 2-2 and Figure 2-3. The e300c1 core version number is 0x8083, and the revision level starts at 0x0010 and changes for each revision of the core. The e300c2 core version number is 0x8084, and the revision level starts at 0x0010. The e300c3 core version number is 0x8085, and the revision level starts at 0x0010.

Table 2-2. Architectural PVR Field Descriptions

Bits	Name	Description
0–15	Version	A 16-bit number that uniquely identifies a particular processor version. This number can be used to determine the version of a processor; it may not distinguish between different end product models if more than one model uses the same processor.
16–31	Revision	A 16-bit number that distinguishes between various releases of a particular version (that is, an engineering change level). The value of the revision portion of the PVR is implementation-specific. The processor revision level is changed for each revision of the device.

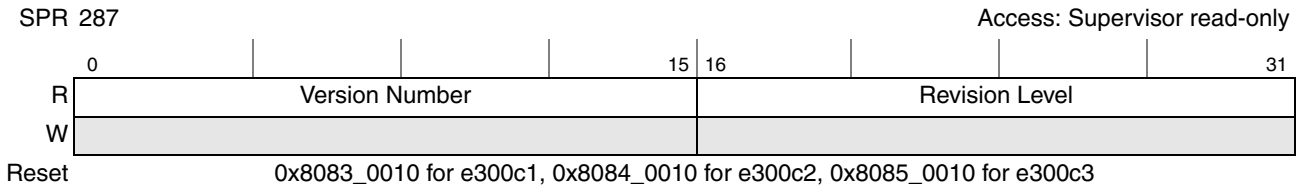


Figure 2-3. e300 Processor Version Register

Table 2-3 describes some of the PVR values for e300-related devices.

Table 2-3. Assigned PVR Values

Device Name	Version No.	Revision No.
MPC603r (PID7)	0x0007	0x1201
G2 core—original	0x0081	0x0011
G2 core	0x8081	0x1010
G2_LE core (general-purpose)	0x8082	0x1010
G2_LE core (general-purpose)	0x8082	0x2010
MPC603e (PID6)	0x0006	0x0101
MPC603e (PID7v)	0x0007	0x0100, 0x0201
e300c1	0x8083	0x0010
e300c2	0x8084	0x0010
e300c3	0x8085	0x0010
Space for future versions		

– Machine state register (MSR). The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsr**), System Call (**sc**), and Return from Interrupt (**rfi**) and Return from Critical Interrupt (**rfci**) instructions. It can be read by the Move from Machine State Register (**mfmsr**) instruction. Figure 2-4 shows the machine state register (MSR).

Access: Supervisor read/write

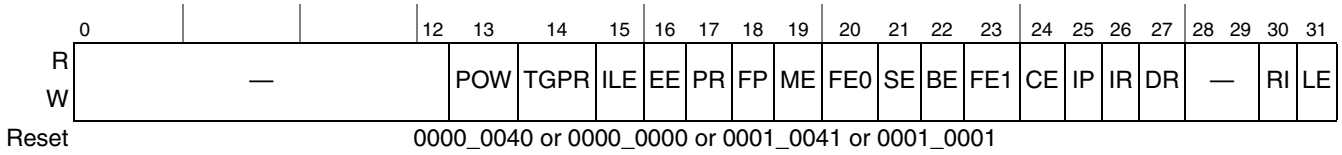


Figure 2-4. Machine State Register

Table 2-4 shows the MSR bit settings.

Table 2-4. MSR Bit Settings

Bits	Name	Description
0[1]	—	Reserved. Full function.
1–4[1]	—	Reserved. Partial function.
5–9[1]	—	Reserved. Full function.
10–12[1]	—	Reserved. Partial function.

Table 2-4. MSR Bit Settings (continued)

Bits	Name	Description
13	POW	Power management enable (implementation-specific) 0 Disables programmable power modes (normal operation mode) 1 Enables programmable power modes (nap, doze, or sleep mode). This bit controls the programmable power modes only; it has no effect on dynamic power management (DPM). MSR[POW] may be altered with an **mtmsr** instruction only. Also, when altering the POW bit, software may alter only this bit in the MSR and no others. The **mtmsr** instruction must be followed by a context-synchronizing instruction. See Chapter 9, "Power Management," for more information.
14	TGPR	Temporary GPR remapping (implementation-specific) 0 Normal operation 1 TGPR mode. GPR0–GPR3 are remapped to TGPR0–TGPR3 for use by TLB miss routines. The contents of GPR0–GPR3 remain unchanged while MSR[TGPR] = 1. Attempts to use GPR4–GPR31 with MSR[TGPR] = 1 yield undefined results. Temporarily replaces TGPR0–TGPR3 with GPR0–GPR3 for use by TLB miss routines. The TGPR bit is set when either an instruction TLB miss, data read miss, or data write miss interrupt is taken. The TGPR bit is cleared by an **rfi** instruction.
15	ILE	Interrupt little-endian mode. When an interrupt occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the interrupt.
16	EE	External interrupt enable 0 The processor ignores external interrupts, system management interrupts, and decrementer interrupts. 1 The processor is enabled to take an external interrupt, system management interrupt, or decrementer interrupt.
17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions 1 The processor can only execute user-level instructions
18	FP	Floating-point available (this bit is read-only on the e300c2 core) 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions and can take floating-point enabled exception type program interrupts.
19	ME	Machine check enable 0 Machine check interrupts are disabled 1 Machine check interrupts are enabled
20	FE0	Floating-point exception mode 0 (see Table 5-9) (This bit is read-only on the e300c2 core)
21	SE	Single-step trace enable 0 The processor executes instructions normally 1 The processor generates a trace interrupt upon the successful completion of the next instruction
22	BE	Branch trace enable 0 The processor executes branch instructions normally 1 The processor generates a trace interrupt upon the successful completion of a branch instruction
23	FE1	Floating-point exception mode 1 (see Table 5-9) (This bit is read-only on the e300c2 core)
24	CE	Critical interrupt enable (e300 implementation-specific) 0 Critical interrupts disabled 1 Critical interrupts enabled; critical interrupt and **rfci** instruction enabled The critical interrupt is an asynchronous implementation-specific interrupt. The critical interrupt vector offset is 0x00A00. The **rfci** instruction is implemented to return from these interrupt handlers. Also, CSRR0 and CSRR1 are used to save and restore the processor state for critical interrupts.

Table 2-4. MSR Bit Settings (continued)

Bits	Name	Description
25	IP	Interrupt prefix. The setting of this bit specifies whether an interrupt vector offset is prepended with Fs or 0s. In the following description, *nnnnn* is the offset of the interrupt. See Table 5-2. 0 Interrupts are vectored to the physical address 0x000*n_nnnn* 1 Interrupts are vectored to the physical address 0xFFF*n_nnnn*
26	IR	Instruction address translation 0 Instruction address translation is disabled 1 Instruction address translation is enabled See Chapter 6, "Memory Management."
27	DR	Data address translation 0 Data address translation is disabled 1 Data address translation is enabled See Chapter 6, "Memory Management."
28–29[1]	—	Reserved. Full function. Bit 29 reserved on e300c1 and e300c2 only.
29	PMM	Performance monitor mark bit (e300c3 only). System software can set PMM when a marked process is running to enable statistics to be gathered only during the execution of the marked process. MSR[PR] and MSR[PMM] together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified in the PMLCa*n*, the state for which monitoring is enabled, counting is enabled.
30	RI	Recoverable interrupt (for system reset and machine check interrupts) 0 Interrupt is not recoverable 1 Interrupt is recoverable
31	LE	Little-endian mode enable 0 The processor runs in big-endian mode 1 The processor runs in little-endian mode. See Section 3.1.2, "Endian Modes and Byte Ordering," for a description of the core operating in true little-endian mode.

[1] All reserved bits should be set to zero for future compatibility.

NOTE

The core defines MSR[13] as the power management enable (POW) bit and MSR[14] as the temporary GPR remapping (TGPR) bit. The e300 allocates MSR[24] is used to enable the critical interrupt and **rfci,** the return from critical interrupt instruction. MSR[31] is used in conjunction with HID2[LET] to indicate the endian mode of operation of the e300 core. These bits are described in Table 2-4.

— Memory management registers:
  – Block-address translation (BAT) registers. The device also supports 16 block-address translation registers (BATs) through the use of 2 independent instruction and data block address translation (IBAT and DBAT) arrays, each containing 8 pairs of BATs, for a total of 16 BAT registers. Effective addresses are compared simultaneously with all eight entries in the BAT array during block translation. Figure 2-1 lists SPR numbers for the BAT registers.
  – SDR1. The SDR1 register specifies the page table base address used in virtual-to-physical address translation. (Note that physical address is referred to as real address in the architecture specification.)

- Segment registers (SRs). The OEA defines sixteen 32-bit segment registers (SR0–SR15). The fields in the segment register are interpreted differently depending on the value of bit 0.

— Interrupt-handling registers
  - Data address register (DAR). After a data access or an alignment interrupt, the DAR is set to the effective address generated by the faulting instruction.
  - The SPRG0–SPRG7 registers are provided for operating system use, which reduce the latency that may be incurred because of saving registers to memory while in a handler and also assist in searching the page tables in software. If software table searching is not enabled, then these registers may be used for any supervisor purpose. Note that the e300 core implements four additional SPRGs (SPRG4–SPRG7) than previous PowerPC cores. These additional registers are not defined by the PowerPC architecture. The format of these registers is defined in Section 2.2.11, "SPRG0–SPRG7."
  - DSISR. The DSISR defines the cause of data access and alignment interrupts.

    Machine status save/restore register [0–1] (SRR0, SRR1). The SRR0 and SRR1 are used to save machine status on interrupts and to restore machine status when an **rfi** instruction is executed. For more information refer to Chapter 6, "Memory Management."

NOTE

The e300 core implements the KEY bit (bit 12) in the SRR1 register to simplify the table search software. For more information refer to Chapter 6, "Memory Management."

Note that to support critical interrupts, two new registers, CSRR0 and CSRR1, are implemented on the e300 core, which are not defined by the PowerPC architecture. These registers have the same bit assignments as SRR0 and SRR1, albeit with different SPR numbers, as described in Section 2.2, "Implementation-Specific Registers."

— Miscellaneous registers:
  - The time base facility (TB) for writing. The TB is a 64-bit register pair that can be used to provide time-of-day or interval timing. It consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). The TB is incremented once every four core input clock cycles.
  - Decrementer (DEC). The DEC register is a 32-bit decrementing counter that provides a mechanism for causing a decrementer interrupt after a programmable delay. The DEC is decremented once every four core input clock cycles.

2.2 Implementation-Specific Registers

This section describes the implementation-specific registers of the e300 core. The core defines the following registers used for software table search operations: DMISS, IMISS, DCMP, ICMP, HASH1, HASH2, and RPA. These registers should be accessed only when address translation is disabled (MSR[IR] and MSR[DR] are both zero). For a complete discussion, refer to Section 6.5.2, "Implementation-Specific Table Search Operation."

The implementation-specific registers also include HID0, HID1, HID2, and IABR SPRs. All of these registers can be accessed by supervisor-level instructions only using the SPR numbers shown in Figure 2-1.

In addition, the e300 core defines the following implementation-specific registers:

- Eight additional BATs (IBAT4–IBAT7 and DBAT4–DBAT7), providing better performance in protecting accesses on a segment, block, or page basis along with memory accesses and I/O accesses. See Figure 2-1 for a list of the SPR numbers for the BAT arrays.
- Two critical interrupt registers (CSRR0, CSRR1), which are implementation-specific. The CSRR0 and CSRR1 registers support the critical interrupt function, which have the same bit assignments as SRR0 and SRR1, respectively. The effective address for resuming program execution is saved into CSRR0 and the content of the MSR is saved into CSRR1. An additional **rfci** instruction is implemented for supporting the return from a critical interrupt, selecting the CSRR0 and CSRR1 registers.
- Four additional interrupt handling SPRG registers, which are provided for operating system use.
- A new system version register (SVR). See Section 2.2.12, “System Version Register (SVR),” for bit definitions.
- System memory base address (MBAR) is a new implementation-specific register for the G2_LE core. It supports a system-level memory map. See Section 2.2.13, “System Memory Base Address (MBAR),” for more information.
- One new instruction address breakpoint control register (IBCR), two new data address breakpoint registers (DABR, DABR2), and one new data address breakpoint control register (DBCR) are implemented in the e300 processor core. All of these new registers, as well as the IABR2 (instruction address breakpoint register 2), are implementation-specific and are described in the Section 2.2.14, “Instruction Address Breakpoint Registers (IABR and IABR2),” and Section 2.2.16, “Data Address Breakpoint Register (DABR and DABR2).”
- Performance monitor registers are available in the e300c3:
  - The performance monitor counter registers (PMC0–PMC3) are 32-bit counters used to count software-selectable events. Each counter counts up to 128 events. UPMC0–UPMC3 provide user-level read access to these registers. Reference events are those that should be applicable to most microprocessor microarchitectures and be of general value. They are identified in Figure 2-1.
  - The performance monitor global control register (PMGC0) controls the counting of performance monitor events. It takes priority over all other performance monitor control registers. UPMGC0 provides user-level read access to PMGC0.
  - The performance monitor local control registers (PMLCa0–PMLCa3) control each individual performance monitor counter. Each counter has a corresponding PMLCa register. UPMLCa0–UPMLCa3 provide user-level read access to PMLCa0–PMLCa3).

2.2.1 Hardware Implementation Register 0 (HID0)

Figure 2-5 shows the e300 implementation of HID0. HID0 can be accessed with **mtspr** and **mfspr** using SPR1008.

SPR 1008 — Access: Supervisor read/write

	0	1	2	3	4	5	6	7	8	9	10	11	12	14	15
R W	EMCP	ECPE	EBA	EBD	SBCLK	—	ECLK	PAR	DOZE	NAP	SLEEP	DPM	—		NHR
Reset	All zeros														

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	ICE	DCE	ILOCK	DLOCK	ICFI	DCFI	—		IFEM	—		FBIOB	ABE	—		NOOPTI
Reset	All zeros															

Figure 2-5. HID0 Register

Table 2-5 shows the bit definitions for HID0.

Table 2-5. e300 HID0 Field Descriptions

Bits	Name	Function
0	EMCP	Enable *mcp*. The purpose of this bit is to mask out machine check interrupts caused by assertion of *mcp*, similar to how MSR[EE] can mask external interrupts. 0 Masks *mcp*. Asserting *mcp* does not generate a machine check interrupt or a checkstop. 1 Asserting *mcp* causes checkstop if MSR[ME] = 0 or a machine check interrupt if ME = 1
1	ECPE	Enable cache parity errors. 0 Disables instruction and data cache parity error reporting 1 Allows a detected cache parity error to cause a machine check interrupt if MSR[ME] = 1 or a checkstop if MSR[ME] = 0
2	EBA	Enable *ap_in[0:3]* and *ape* for address parity checking. 0 Disables address parity checking during a snoop operation 1 Allows an address parity error during snoop operations to cause a checkstop if MSR[ME] = 0 or a machine check interrupt if MSR[ME] = 1
3	EBD	Enable *dpe* for data parity checking. 0 Disables data parity checking 1 Allows a data parity error during reads to cause a checkstop if MSR[ME] = 0 or a machine check interrupt if MSR[ME] = 1
4	SBCLK	*clk_out* output enable. Used in conjunction with HID0[ECLK] and *hreset* to configure *clk_out.*
5	—	Reserved, should be cleared
6	ECLK	*clk_out* output enable. Used in conjunction with HID0[SBCLK] and the *hreset* signal to configure *clk_out.*
7	PAR	Disable precharge of *artry_out* 0 Precharge of *artry_out* enabled 1 Alters bus protocol slightly by preventing the processor from driving *artry_out* to high (negated) state. If this is done, the integrated device must restore the signals to the high state.

Table 2-5. e300 HID0 Field Descriptions (continued)

Bits	Name	Function
8	DOZE	Doze mode enable. Operates in conjunction with MSR[POW]. 0 Doze mode disabled 1 Doze mode enabled. Doze mode is invoked by setting MSR[POW] while this bit is set. In doze mode, the PLL, time base, and snooping remain active.
9	NAP	Nap mode enable. Operates in conjunction with MSR[POW]. The $\overline{qreq}$ signal is asserted to indicate that the processor is ready to enter nap mode. If the system logic determines that the processor may enter nap mode, the quiesce acknowledge signal, $\overline{qack}$, is asserted to notify the processor. 0 Nap mode disabled 1 Nap mode enabled. Nap mode is invoked by setting MSR[POW] while this bit is set. In nap mode, the PLL and time base remain active.
10	SLEEP	Sleep mode enable. Operates in conjunction with MSR[POW]. 0 Sleep mode disabled 1 Sleep mode enabled. Sleep mode is invoked by setting MSR[POW] while this bit is set. $\overline{qreq}$ is asserted to indicate that the processor is ready to enter sleep mode. If the system logic determines that the processor may enter sleep mode, the quiesce acknowledge signal, $\overline{qack}$, is asserted back to the processor. Once $\overline{qack}$ assertion is detected, the processor enters sleep mode after several processor clocks. At this point, the system logic may turn off the PLL by first configuring *pll_cfg[0:6]* to PLL bypass mode, then disabling *sysclk*.
11	DPM	Dynamic power management enable 0 Dynamic power management is disabled 1 Functional units enter a low-power mode automatically if the unit is idle. This does not affect operational performance and is transparent to software or any external hardware.
12–15	—	Reserved, should be cleared.
16	ICE	Instruction cache enable 0 The instruction cache is neither accessed nor updated. All pages are accessed as if they were marked cache-inhibited (WIM = x1x). Potential cache accesses from the bus (snoop and cache operations) are ignored. In the disabled state for the L1 caches, the cache tag state bits are ignored and all instruction fetches are propagated to the coherent system bus (CSB) as single-beat or burst transactions, depending on the value of HID2[IFEB]. For those transactions, however, $\overline{ci}$ reflects the state of the I bit in the MMU for that page regardless of cache disabled status. ICE is zero at power-up. 1 The instruction cache is enabled
17	DCE	Data cache enable 0 The data cache is neither accessed nor updated. All pages are accessed as if they were marked cache-inhibited (WIM = x1x). Potential cache accesses from the bus (snoop and cache operations) are ignored. In the disabled state for the L1 caches, the cache tag state bits are ignored and all data read and write accesses are propagated to the CSB as single-beat transactions. For those transactions, however, $\overline{ci}$ reflects the state of the I bit in the MMU for that page regardless of cache disabled status. DCE is zero at power-up. 1 The data cache is enabled
18	ILOCK	Instruction cache lock 0 Normal operation 1 The entire instruction cache is locked (that is, all eight ways of the cache are locked). A locked cache supplies data normally on a hit, but the access is treated as a cache-inhibited transaction on a miss. On a miss, the transaction to the bus is single-beat or burst, depending on the value of HID2[IFEB]; however, $\overline{ci}$ still reflects the state of the I bit in the MMU for that page, regardless of whether the cache is locked or disabled. To prevent locking during a cache access, an **isync** instruction must precede the setting of ILOCK.

Table 2-5. e300 HID0 Field Descriptions (continued)

Bits	Name	Function
19	DLOCK	Data cache lock 0 Normal operation 1 The entire data cache is locked (that is, all eight ways of the cache are locked). A locked cache supplies data normally on a hit, but is treated as a cache-inhibited transaction on a miss. On a miss, the transaction to the bus is single-beat; however, $\overline{ci}$ still reflects the state of the I bit in the MMU for that page regardless of whether the cache is locked or disabled. A snoop hit to a locked L1 data cache performs as if the cache were not locked. A cache block invalidated by a snoop remains invalid until the cache is unlocked. To prevent locking during a cache access, a **sync** instruction must precede the setting of DLOCK.
20	ICFI	Instruction cache flash invalidate 0 The instruction cache is not invalidated. The bit is cleared when the invalidation operation begins (usually the next cycle after the write operation to the register). The instruction cache must be enabled for the invalidation to occur. 1 An invalidate operation is issued that marks the state of each instruction cache block as invalid. Cache access is blocked during this time. Setting ICFI clears all the valid bits of the blocks and the PLRU bits to point to way L0 of each set. For the e300 core, the proper use of the ICFI and DCFI bits is to set and clear them with two consecutive **mtspr** operations.
21	DCFI	Data cache flash invalidate 0 The data cache is not invalidated. The bit is cleared when the invalidation operation begins (usually the next cycle after the write operation to the register). The data cache must be enabled for the invalidation to occur. 1 An invalidate operation is issued that marks the state of each data cache block as invalid without writing back modified cache blocks to memory. Cache access is blocked during this time. Bus accesses to the cache are signaled as a miss during invalidate-all operations. Setting DCFI clears all the valid bits of the blocks and the PLRU bits to point to way L0 of each set. For the e300 core, the proper use of the ICFI and DCFI bits is to set and clear them with two consecutive **mtspr** operations.
22–23	—	Reserved, should be cleared.
24	IFEM	Enable M bit on bus for instruction fetches 0 M bit not reflected on bus for instruction fetches. Instruction fetches are treated as nonglobal on the bus. 1 Instruction fetches reflect the M bit from the WIM settings
25	DECAREN	Decrementer auto reload (not supported on the e300c1) 0 Normal operation. 1 Decrementer loads last mtdec value for precise periodic interrupt.
25–26	—	Reserved, should be cleared. Bit 25 reserved in e300c1 only.
27	FBIOB	Force branch indirect on the bus 0 Register indirect branch targets are fetched normally 1 Forces register indirect branch targets to be fetched externally
28	ABE	Address broadcast enable. Controls whether certain address-only operations (such as cache operations) are broadcast on the bus. 0 Address-only operations affect only local caches and are not broadcast 1 Address-only operations are broadcast on the bus Affected instructions are **dcbi**, **dcbf**, and **dcbst**. Note that these cache control instruction broadcasts are not snooped by the e300 core. Refer to Section 4.3.3, “Data Cache Control,” for more information.

Table 2-5. e300 HID0 Field Descriptions (continued)

Bits	Name	Function
29–30	—	Reserved, should be cleared.
31	NOOPTI	No-op the data cache touch instructions 0 The **dcbt** and **dcbtst** instructions are enabled 1 The **dcbt** and **dcbtst** instructions are no-oped internal to the e300 core

Table 2-6 shows how HID0[SBCLK], HID0[ECLK], and $\overline{hreset}$ are used to configure *clk_out*.

Table 2-6. HID0[SBCLK] and HID0[ECLK] *clk_out* Configuration

$\overline{hreset}$	HID0[ECLK]	HID0[SBCLK]	*clk_out*
Asserted	x	x	Core
Negated	0	0	Core
Negated	0	1	Core clock frequency/2
Negated	1	0	Core
Negated	1	1	Bus

HID0 can be accessed with **mtspr** and **mfspr** using SPR1008.

2.2.2 Hardware Implementation Register 1 (HID1)

The e300 implementation of HID1 is shown in Figure 2-6.

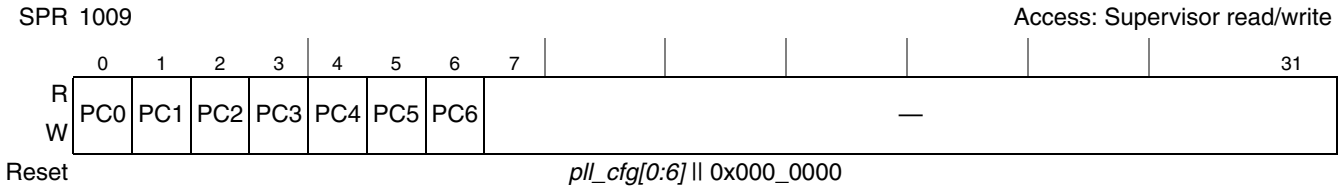


Figure 2-6. HID1 Register

Table 2-7 shows the bit definitions for HID1.

Table 2-7. HID1 Bit Settings

Bits	Name	Description
0	PC0	PLL configuration bit 0 (read-only)
1	PC1	PLL configuration bit 1 (read-only)
2	PC2	PLL configuration bit 2 (read-only)
3	PC3	PLL configuration bit 3 (read-only)
4	PC4	PLL configuration bit 4 (read-only)
5	PC5	PLL configuration bit 5 (read-only)

Table 2-7. HID1 Bit Settings (continued)

Bits	Name	Description
6	PC6	PLL configuration bit 6 (read-only)
7–31	—	Reserved, should be cleared

Note: The clock configuration bits reflect the state of the *pll_cfg[0:6]* signals.

HID1 can be accessed with **mfspr** using SPR1009.

2.2.3 Hardware Implementation Register 2 (HID2)

The core implements an additional hardware implementation-dependent HID2 register, shown in Figure 2-7, which enables cache way-locking; the HID2 also enables true little-endian mode and the new additional BAT registers. It is a supervisor-only, read/write, implementation-specific special-purpose register (SPR) which is accessed as SPR1011 (decimal). The HID2 bits are shown in Table 2-12.

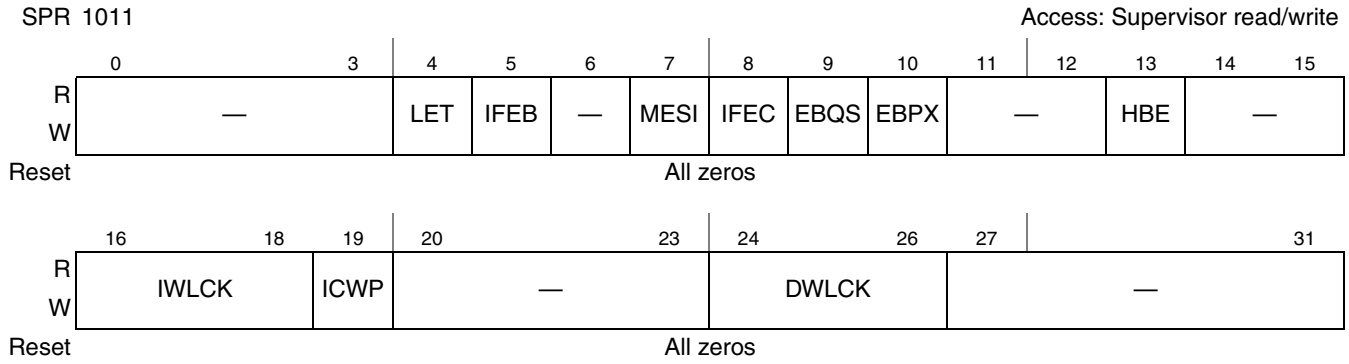


Figure 2-7. HID2 Register

Table 2-8 describes the HID2 fields.

Table 2-8. e300 HID2 Field Descriptions

Bits	Name	Description
0–3	—	Reserved, should be cleared.
4	LET	True little-endian. This bit enables true little-endian mode operation for instruction and data accesses. This bit is set to reflect the state of the *tle* signal at the negation of $\overline{hreset}$. This bit is used in conjunction with MSR[LE] to determine the endian mode of operation. 0 Modified (PowerPC) little-endian mode, not supported in the e300 core. 1 True little-endian mode, when MSR[LE] = 1 Changing the value of this bit during normal operation is not recommended
5	IFEB	Instruction fetch burst extension. This bit enables the instruction fetch burst extension. 0 Instruction fetch burst extension disabled 1 Instruction fetch burst extension enabled
6	—	Reserved, should be cleared.

Table 2-8. e300 HID2 Field Descriptions (continued)

Bits	Name	Description
7	MESISTATE	MESI state enable. This bit enables the four-state MESI cache coherency protocol. 0 MESI disabled. The data cache uses a three-state MEI coherency protocol. 1 MESI enabled. The data cache uses a four-state MESI protocol.
8	IFEC	Instruction fetch cancel extension. This bit enables the instruction fetch cancel extension. 0 Instruction fetch cancel extension disabled 1 Instruction fetch cancel extension enabled
9	EBQS	Enable BIU queue sharing. This bit enables data cache queue sharing. 0 Data cache queue sharing disabled 1 Data cache queue sharing enabled
10	EBPX	Enable BIU pipeline extension.This bit enables the bus interface unit pipeline extension. 0 BIU pipeline extension disabled; 1 level pipeline 1 BIU pipeline extension enabled; 1-1/2 level pipeline
11–12	—	Reserved for e300c1, should be cleared.
11	ELRW	Enable weighted LRU. This bit enables the use of an adjusted (weighted) LRU. 0 Normal operation. 1 The dcbt, dcbtst, and dcbz instructions use and adjusted (weighted) LRU such that they always select and replace the lowest unlocked way in the data cache.
12	NOKS	Reserved for e300c1, should be cleared. For e300c2, e300c3: No kill for snoop. This bit enables the forcing of kill-type snoops to flush data instead of killing it. 0 Normal operation. 1 Forces write-with-kill snoops to flush instead of kill (snoop can never kill data).
13	HBE	High BAT enable. Regardless of the setting of HID2[HBE], these BATs are accessible by **mfspr** and **mtspr**. 0 IBAT[4–7] and DBAT[4–7] are disabled 1 IBAT[4–7] and DBAT[4–7] are enabled
14–15	—	Reserved, should be cleared.
16–18	IWLCK[0–2]	Instruction cache way-lock. Useful for locking blocks of instructions into the instruction cache for time-critical applications that require deterministic behavior. 000 = no ways locked 001 = way 0 locked 010 = way 0 through way 1 locked 011 = way 0 through way 2 locked 100 = way 0 through way 3 locked in e300c1. Way 0 through way 2 locked in e300c2, e300c3. 101 = way 0 through way 4 locked in e300c1. Way 0 through way 2 locked in e300c2, e300c3. 110 = way 0 through way 5 locked in e300c1. Way 0 through way 2 locked in e300c2, e300c3. 111 = way 0 through way 6 locked in e300c1. Way 0 through way 2 locked in e300c2, e300c3. Setting HID0[ILOCK] will lock all ways in all e300 cores.
19	ICWP	Instruction cache way protection. Used to protect locked ways in the instruction cache from being invalidated. 0 Instruction cache way protection disabled 1 Instruction cache way protection enabled
20–23	—	Reserved, should be cleared.

Table 2-8. e300 HID2 Field Descriptions (continued)

Bits	Name	Description
24–26	DWLCK[0–2]	Data cache way-lock. Useful for locking blocks of data into the data cache for time-critical applications where deterministic behavior is required. 000 = no ways locked 001 = way 0 locked 010 = way 0 through way 1 locked 011 = way 0 through way 2 locked 100 = way 0 through way 3 locked in e300c1. Way 0 through way 2 locked in e300c2, e300c3. 101 = way 0 through way 4 locked in e300c1. Way 0 through way 2 locked in e300c2, e300c3. 110 = way 0 through way 5 locked in e300c1. Way 0 through way 2 locked in e300c2, e300c3. 111 = way 0 through way 6 locked in e300c1. Way 0 through way 2 locked in e300c2, e300c3. Setting HID0[DLOCK] will lock all ways in all e300cores.
27–31	—	Reserved, should be cleared.

2.2.4 Data and Instruction TLB Miss Address Registers (DMISS and IMISS)

DMISS and IMISS, shown in Figure 2-8, are loaded automatically on a data or instruction TLB miss. DMISS and IMISS contain the effective address of the access that caused the TLB miss interrupt. The contents are used by the core when calculating the values of HASH1 and HASH2 and by the **tlbld** and **tlbli** instructions when loading a new TLB entry. Note that the core always loads DMISS with a big-endian address, even when MSR[LE] is set. These registers are both read- and write-accessible. However, caution should be used when writing to these registers.

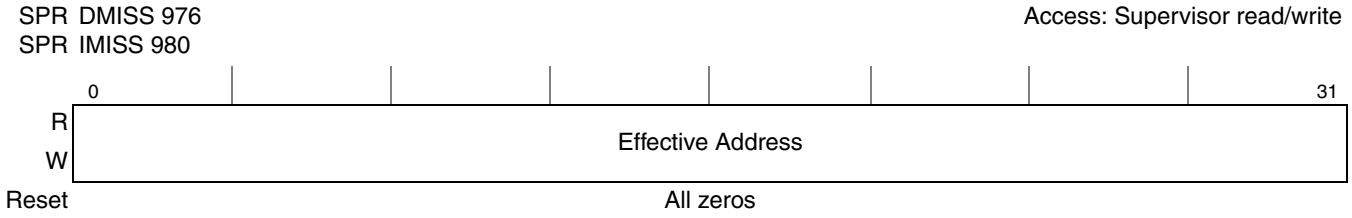


Figure 2-8. DMISS and IMISS Registers

2.2.5 Data and Instruction TLB Compare Registers (DCMP and ICMP)

DCMP and ICMP, shown in Figure 2-9, contain the first word in the required PTE. The contents are constructed automatically from the contents of the segment registers and the effective address (DMISS or IMISS) when a TLB miss interrupt occurs. Each PTE read from the tables during the table search process should be compared with this value to determine if the PTE is a match. Upon execution of a **tlbld** or **tlbli** instruction, the upper 25 bits of the DCMP or ICMP register and 11 bits of the effective address are loaded into the first word of the selected TLB entry. These registers are read and write to the software.

SPR 977 (DCMP)
SPR 981 (ICMP)

Access: Supervisor read/write

Figure 2-9. DCMP and ICMP Registers

Table 2-9 describes the bit settings for the DCMP and ICMP registers.

Table 2-9. DCMP and ICMP Bit Settings

Bits	Name	Description
0	V	Valid bit. Set by the processor on a TLB miss interrupt.
1–24	VSID	Virtual segment ID. Copied from VSID field of corresponding segment register.
25	H	Hash function identifier. Cleared by the processor on a TLB miss interrupt.
26–31	API	Abbreviated page index. Copied from API of effective address.

2.2.6 Primary and Secondary Hash Address Registers (HASH1 and HASH2)

HASH1 and HASH2, shown in Figure 2-10, contain the physical addresses of the primary and secondary PTEGs, respectively, for the access that caused the TLB miss interrupt. For convenience, the device automatically constructs the full physical address by routing SDR1 bits 0–6 into HASH1 and HASH2 and clearing the lower 6 address bits. These read-only registers are constructed from the DMISS or IMISS contents (the register choice is determined by which miss most recently occurred).

SPR 978 (HASH1)
SPR 979 (HASH2)

Access: Supervisor read-only

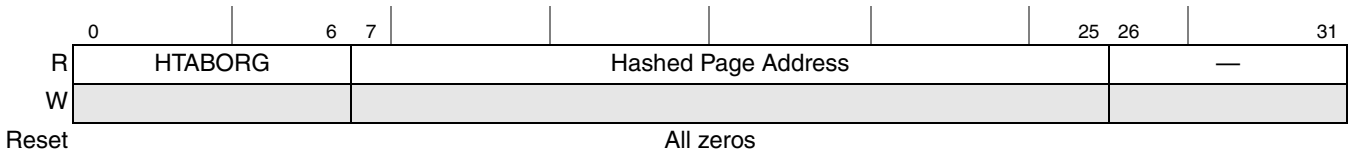


Figure 2-10. HASH1 and HASH2 Registers

Table 2-10 describes the bit settings of the HASH1 and HASH2 registers.

Table 2-10. HASH1 and HASH2 Bit Settings

Bits	Name	Description
0–6	HTABORG	Copy of the upper 7 bits of the HTABORG field from SDR1
7–25	Hashed page address	Address bits 7–25 of the PTEG to be searched
26–31	—	Reserved

2.2.7 Required Physical Address Register (RPA)

During a page table search operation, the software must load the RPA, shown in Figure 2-11, with the second word of the correct PTE. When the **tlbld** or **tlbli** instruction is executed, the RPA and DMISS or IMISS register are merged and loaded into the selected TLB entry. The referenced (R) bit is ignored when the write occurs (no location exists in the TLB entry for this bit). The RPA register is read- and write-accessible to the software.

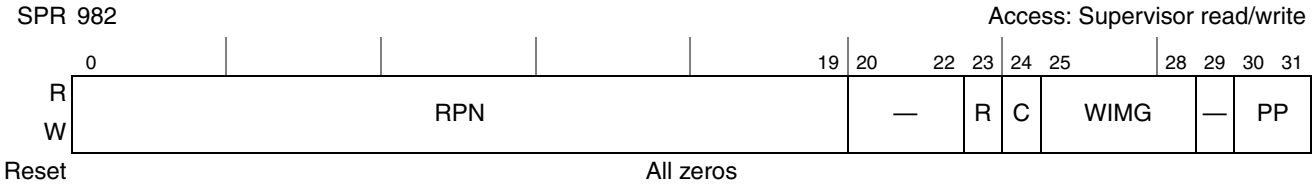


Figure 2-11. Required Physical Address Register (RPA)

Table 2-11 describes the bit settings of the RPA register.

Table 2-11. RPA Bit Settings

Bits	Name	Description
0–19	RPN	Physical page number from PTE
20–22	—	Reserved
23	R	Referenced bit from PTE
24	C	Changed bit from PTE
25–28	WIMG	Memory/cache access attribute bits
29	—	Reserved
30–31	PP	Page protection bits from PTE

2.2.8 BAT Registers (BAT4–BAT7)

The MMU has four additional IBAT and four additional DBAT array entries that provide a mechanism for translating additional blocks as large as 256 Mbytes from the 32-bit effective address space into the physical memory space. This can be used for translating large address ranges whose mappings do not change frequently.

BATs are software-controlled arrays that store the available block address translations on-chip. The core supports block address translation through the use of two independent instruction and data block address translation (IBAT and DBAT) arrays; each array is comprised of four additional entries used for instruction accesses and four additional entries used for data accesses.

IBAT4–IBAT7 and DBAT4–DBAT7 are implementation-specific registers on the e300 core, which are optionally enabled in HID2. The format of these registers is the same as that of IBAT0–IBAT3 and DBAT0–DBAT3. Each BAT array entry consists of a pair of BAT registers—an upper and a lower BAT register for each entry. Figure 2-12 and Figure 2-13 show the format and bit definitions of the upper and lower BATs for 32-bit processor cores, respectively.

SPR 528 (IBAT0U) 536 (DBAT0U) Access: Supervisor read/write
530 (IBAT1U) 538 (DBAT1U)
532 (IBAT2U) 540 (DBAT2U)
534 (IBAT3U) 542 (DBAT3U)
560 (IBAT4U) 568 (DBAT4U)
562 (IBAT5U) 570 (DBAT5U)
564 (IBAT6U) 572 (DBAT6U)
566 (IBAT7U) 574 (DBAT7U)

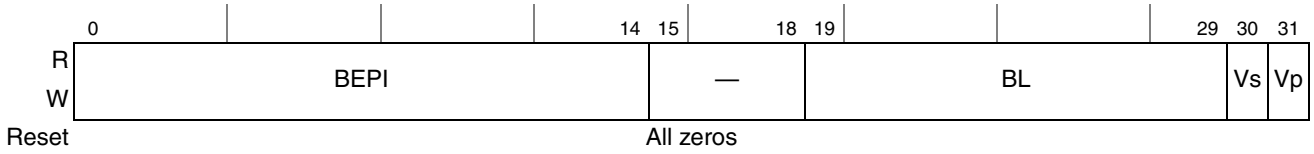
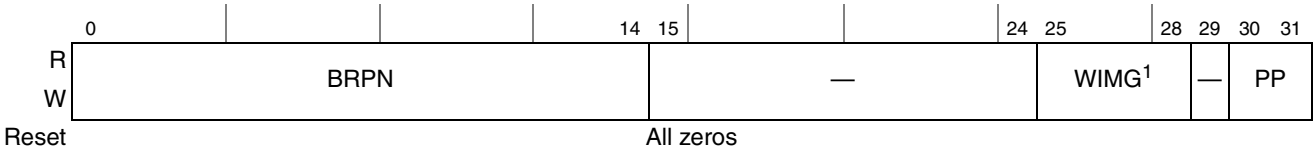


Figure 2-12. Upper BAT Register

SPR 529 (IBAT0L) 537 (DBAT0L) Access: Supervisor read/write
531 (IBAT1L) 539 (DBAT1L)
533 (IBAT2L) 541 (DBAT2L)
535 (IBAT3L) 543 (DBAT3L)
561 (IBAT4L) 569 (DBAT4L)
563 (IBAT5L) 571 (DBAT5L)
565 (IBAT6L) 573 (DBAT6L)
567 (IBAT7L) 575 (DBAT7Ul)



1 W and G bits are not defined for IBAT registers. Attempting to write to these bits causes boundedly-undefined results.

Figure 2-13. Lower BAT Register

The BAT registers contain the effective-to-physical address mappings for blocks of memory. This mapping includes the effective address bits that are compared with the effective address of the access, the memory/cache access mode bits (WIMG), and the protection bits for the block. The size of the block and the starting address of the block are defined by the physical block number (BRPN) and block size mask (BL) fields.

The sixteen new BAT registers are enabled by HID2[HBE]. However, regardless of the setting of this bit, the BAT registers are accessible by the **mfspr** and **mtspr** instructions and are only accessible to supervisor-level programs. See Section 2.2.3, “Hardware Implementation Register 2 (HID2),” for more information on the HBE bit.

2.2.9 Critical Interrupt Save/Restore Register 0 (CSRR0)

CSRR0 is used to save the return address of critical interrupts. It is set to the effective address of the instruction that the processor would have attempted to complete next if no critical interrupt had occurred. The format of CSRR0 is shown in Figure 2-14.

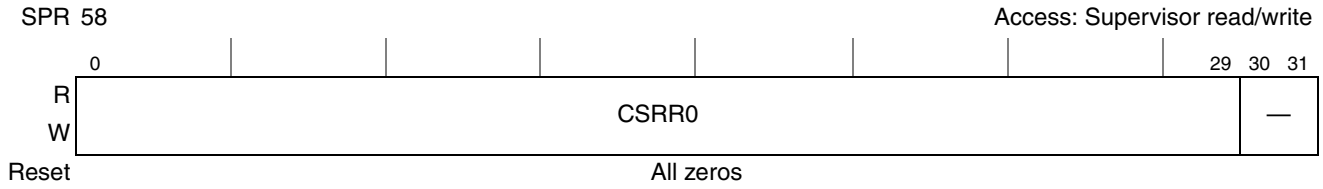


Figure 2-14. Critical Interrupt Save/Restore Register 0 (CSRR0)

For information on how specific interrupts affect CSRR0, refer to the descriptions of individual interrupts in Chapter 5, “Interrupts and Exceptions.”

2.2.10 Critical Interrupt Save/Restore Register 1 (CSRR1)

CSRR1 is used to save machine status on interrupts and to restore machine status when an **rfci** instruction is executed. Figure 2-15 shows the CSRR1 format.

Figure 2-15. Critical Interrupt Save/Restore Register 1 (CSRR1)

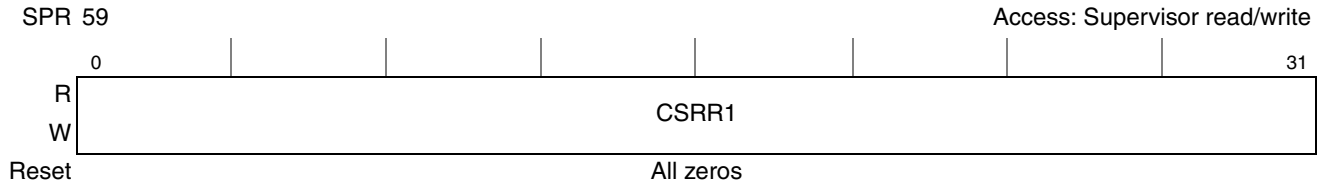


Figure 2-16. Critical Interrupt Save/Restore Register 0 (CSRR0)

For information on how specific interrupts affect CSRR1, refer to the individual interrupts in Chapter 5, “Interrupts and Exceptions.”

2.2.11 SPRG0–SPRG7

The core provides four additional SPRG (SPRG4–SPRG7) registers for general operating system use, such as performing a fast state save or for supporting multiprocessor implementations. The formats of the SPRG registers are shown in Figure 2-17. Note that SPRG4-SPRG7 are not supported in the G2 core.

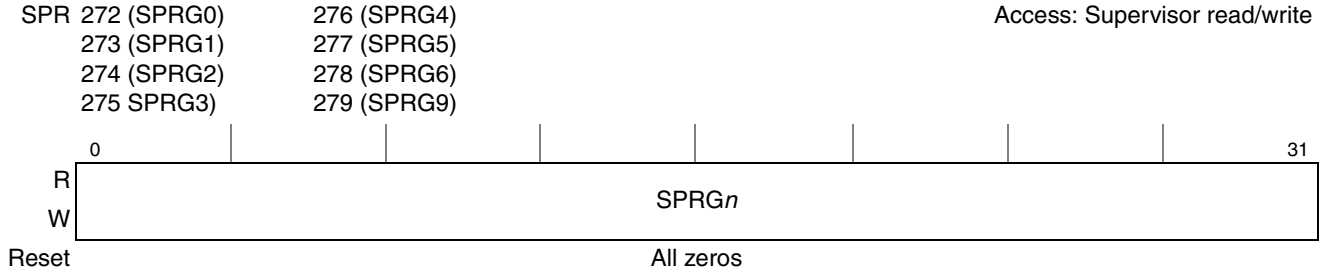


Figure 2-17. SPRG*n* Register

For information on conventional uses for SPRG4–SPRG7, refer to Section 5.2.1.3, “SPRG0–SPRG7.”

2.2.12 System Version Register (SVR)

The system version register (SVR) is a 32-bit read-only register that identifies the specific version (model) and revision level of the system on a chip (SoC). Supervisor mode write access is reserved for future use. Figure 2-18 shows an implementation of the SVR, although it should be noted that this register is determined by the SoC.

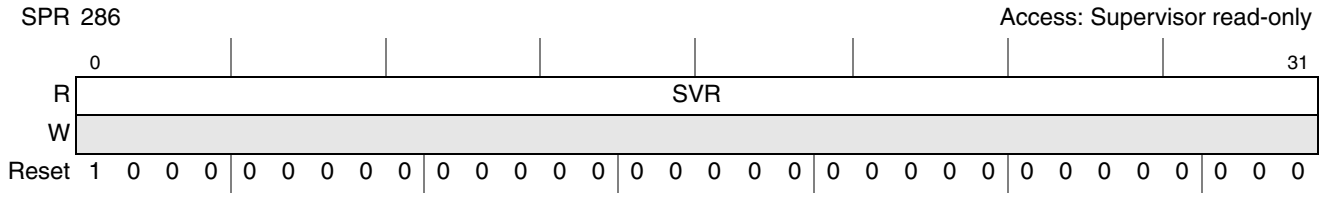


Figure 2-18. SVR Register

The SVR can be accessed with **mfspr** using SPR286. The bits in SVR are defined in Table 2-12.

Note that all bits within this register are guaranteed to be configured by the SOC and unused bits are cleared to zero. Also, SVR4–SVR15 are control fields for this register.

Table 2-12. System Version Register (SVR) Bit Settings

Bits	Name	Description
0–3	CID	Company or manufacturer ID. These bits are required. Bit 0 must set to 1.
4–15	SID[1]	SoC ID. This required field is used to identify the SoC device.
16–19	PROC	Process revision field. This optional field is used to indicate different process revisions of the SoC.
20–23	MFG	Manufacturing revision. This optional field identifies uniquely different manufacturing revisions of the SoC.
24–27	MJREV	Major SoC design revision indicator. This is a required field.
28–31	MNREV	Minor SoC design revision indicator. This is a required field.

[1] The SoC value is an optional field assigned by the SoC design integrator.

2.2.13 System Memory Base Address (MBAR)

The core implements a new memory base address register (MBAR) to support the system level memory map. The MBAR can be accessed with **mtspr** or **mfspr** using SPR311 in supervisor mode. The present memory base address for the system memory map is stored in this register. It is important to ensure that the present value of the base offset is current in the system memory.

2.2.14 Instruction Address Breakpoint Registers (IABR and IABR2)

The IABR, shown in Figure 2-19, controls the instruction address breakpoint interrupt. In the core, an additional address breakpoint register (IABR2) is implemented. IABR[CEA] holds an effective address to which each instruction's address is compared. The interrupt is enabled by setting IABR[BE]. The interrupt is taken when there is an instruction address breakpoint match on the next instruction to complete. The instruction tagged with the match cannot complete before the breakpoint interrupt is taken. The address of

the instruction which matches the breakpoint condition is stored in SRR0. The tagged instruction is completed and retired on return from the interrupt (**rfi** or **rfci**). The results are then committed to the destination registers and address.

Note that if IABR/IABR2 values are set to any interrupt vector, an unrecoverable processor state occurs.

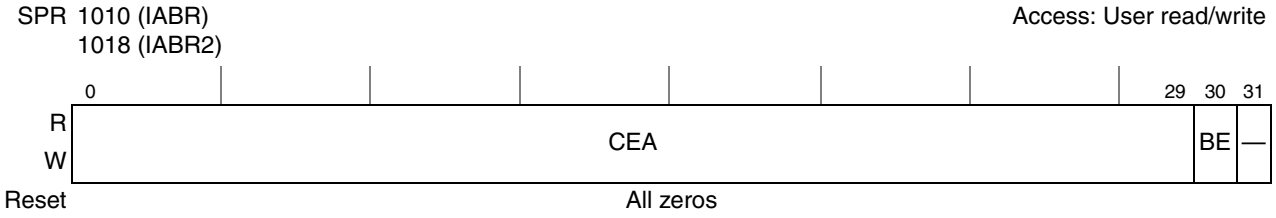


Figure 2-19. IABR and IABR2 Registers

The bits in the IABR and IABR2 are defined in Table 2-13. For more information about the instruction breakpoint interrupt, see Section 5.5.17, “Instruction Address Breakpoint Interrupt (0x01300).”

Table 2-13. Instruction Address Breakpoint Register (IABR and IABR2) Bit Settings

Bits	Name	Description
0–29	CEA	Compare effective address. Word address to be compared.
30	BE	Breakpoint enable. IABR (or IABR2) enabled. Setting this bit enables the IABR interrupt.
31	—	Reserved

2.2.15 Instruction Address Breakpoint Control Register (IBCR)

The IBCR, shown in Figure 2-20, is a supervisor-level register with SPR309 on the e300 core, which is accessible only by using an **mtspr** or **mfspr** instruction. The IBCR controls the compare and match type conditions for IABR and IABR2. Note that IABR and IABR2 must be enabled before the effects of IBCR are realized.

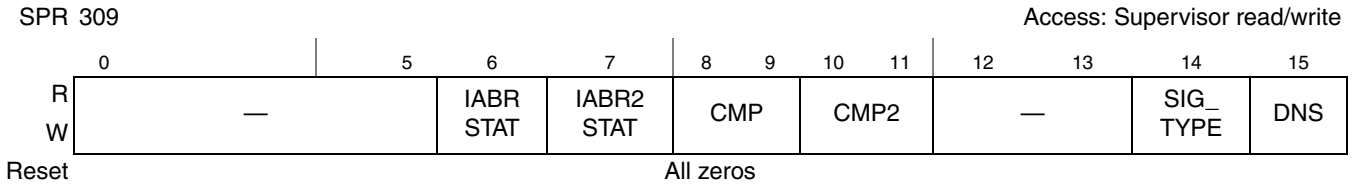


Figure 2-20. IBCR Register

Table 2-14 describes the IBCR fields

Table 2-14. Instruction Address Breakpoint Control Registers (IBCR)

Bits	Name	Description
0–5	—	Reserved
6	IABRSTAT	IABR status. 0 Match on IABR has not occurred. 1 Match on IABR has occurred.

Table 2-14. Instruction Address Breakpoint Control Registers (IBCR) (continued)

Bits	Name	Description
7	IABR2STAT	IABR2 status. 0 Match on IABR2 has not occurred 1 Match on IABR2 has occurred
8–9	CMP	IABR breakpoint compare type 00 Match if instruction's EA equals IABR[CEA] 01 Reserved 10 Match if instruction's EA is less than IABR[CEA] 11 Match if instruction's EA is greater than or equal to IABR[CEA]
10–11	CMP2	IABR2 breakpoint compare type 00 Match if instruction's EA equals IABR2[CEA] 01 Reserved 10 Match if instruction's EA less than IABR2[CEA] 11 Match if instruction's EA greater than or equal to IABR2[CEA]
12	—	Reserved
13	—	Reserved
14	SIG_TYPE	Combinational signal type 0 Instruction's EA matches IABR[CEA] OR instruction's EA matches IABR2[CEA] 1 Instruction's EA matches IABR[CEA] AND instruction's EA matches IABR2[CEA]
15	DNS	Do not signal. Disable $\overline{iabr}$ and $\overline{iabr2}$ output signals 0 Allow signal to toggle on a match 1 Do not toggle signal on match

2.2.16 Data Address Breakpoint Register (DABR and DABR2)

The optional data address breakpoint facility on the e300 core is controlled by optional SPRs, DABR and DABR2. The data address breakpoint facility provides a means to detect data accesses to a designated double-word address. The breakpoint address is compared to the effective address of all data accesses; it does not apply to instruction fetches.

DABR and DABR2, the two data address breakpoint registers shown in Figure 2-21, can both cause the data address breakpoint interrupt.

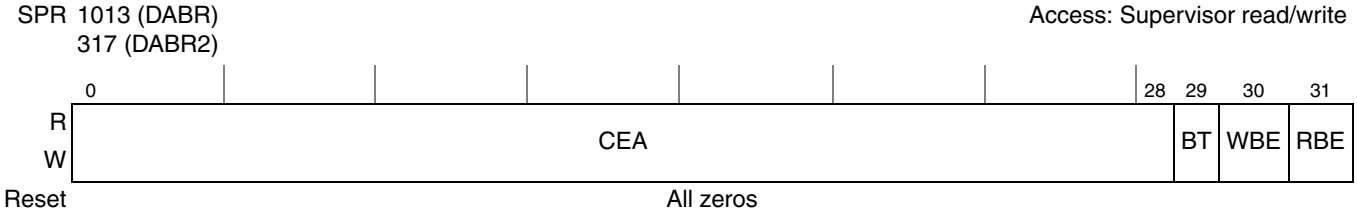


Figure 2-21. DABR and DABR2 Registers

When an enabled data breakpoint condition matches with the address of a data access, a DSI interrupt occurs. When a DSI interrupt is taken to indicate a data breakpoint condition, DAR is set to the data address that causes the breakpoint and DSISR[9] is set. The address of the instruction associated with the breakpoint condition is stored in SRR0.

Note that if the DABR/DABR2 register values are set to match on any interrupt vector, an indeterminate or unrecoverable processor state may occur.

Table 2-15 describes the fields in DABR and DABR2.

Table 2-15. Data Address Breakpoint Registers (DABR and DABR2) Bit Settings

Bits	Name	Description
0–28	CEA	Data address breakpoint
29	BT	Breakpoint translation enable. Match if MSR[DR] = DABR[BT].
30	WBE	Data write enable. Matching on data writes enabled.
31	RBE	Data read enable. Matching on data reads enabled.

A data address breakpoint match is detected for a load or store instruction if the following conditions are met for any byte accessed:

- EA0–EA28 = DABR[CEA]
- MSR[DR] = DABR[BT]
- The instruction is a store and DABR[WBE] = 1 or the instruction is a load and DABR[RBE] = 1

Even if the above conditions are satisfied, it is undefined whether a match occurs in the following cases:

- A store conditional indexed instruction (**stwcx.**) in which the store is not performed
- A load or store string instruction (**lswx** or **stswx**) with a zero length
- A **dcbz**, **dcba**, **eciwx**, or **ecowx** instruction. For the purpose of determining whether a match occurs, **eciwx** is treated as a load and **dcbz**, **dcba**, and **ecowx** are treated as stores. Note that **eciwx** and **ecowx** may generate illegal transactions in some implementations and are not supported in the e300.

The cache management instructions other than **dcbz** and **dcba** never cause a match. If **dcbz** or **dcba** causes a match, some or all of the target memory locations may have been updated.

When a match occurs, a DSI interrupt is generated. Refer to Section 5.5.3, “DSI Interrupt (0x00300),” more information on the data address breakpoint facility.

2.2.17 Data Address Breakpoint Control Register (DBCR)

The DBCR is a supervisor-level register with SPR310 on the e300 core, which is accessible only by using **mtspr** and **mfspr**. The DBCR controls the compare and match type conditions for DABR and DABR2. Figure 2-22 shows the format of the DBCR.

SPR 310 Access: Supervisor read/write

	0	5	6	7	8	9	10	11	12	13	14	15
R W	—		DABR STAT	DABR2 STAT	CMP		CMP2		—		SIG_ TYPE	DNS
Reset	All zeros											

Figure 2-22. DBCR Register

Table 2-16 provides the description of DBCR bit settings.

Table 2-16. Data Address Breakpoint Control Registers (DBCR)

Bits	Name	Description
0–5	—	Reserved
6	DABRSTAT	DABR status 0 Match on DABR has not occurred 1 Match on DABR has occurred
7	DABR2STAT	DABR2 status 0 Match on DABR2 has not occurred 1 Match on DABR2 has occurred
8–9	CMP	DABR breakpoint compare type 00 Match if data's EA equals DABR[CEA] 01 Reserved 10 Match if data's EA less than DABR[CEA] 11 Match if data's EA greater than or equal to DABR[CEA]
10–11	CMP2	DABR2 breakpoint compare type 00 Match if data's EA equals DABR2[CEA] 01 Reserved 10 Match if data's EA less than DABR2[CEA] 11 Match if data's EA greater than or equal to DABR2[CEA]
12–13	—	Reserved
14	SIG_TYPE	Combinational signal type 0 Data access EA matches DABR[CEA] OR EA matches DABR2[CEA] 1 Data access EA matches DABR[CEA] AND EA matches DABR2[CEA]
15	DNS	Do not signal. Disable $\overline{dabr}$ and $\overline{dabr2}$ output signals. 0 Allow signal to toggle on a match 1 Do not toggle signal on match
16–31	—	Reserved

2.2.18 Performance Monitor Registers

The performance monitor provides a set of PMRs for defining, enabling, and counting conditions that trigger the performance interrupt. It also the performance monitor interrupt vector.

The supervisor-level performance monitor registers are accessed with **mtpmr** and **mfpmr**. Attempting to read or write supervisor-level registers in user-mode causes a privilege exception.

The user-level performance monitor registers are read-only and are accessed with the **mfpmr** instruction. Attempting to write these user-level registers in either supervisor or user mode causes an illegal instruction exception.

Chapter 11, “Performance Monitor for a detailed description of performance monitor registers.

Chapter 3
Instruction Set Model

This chapter describes the operand conventions as they are represented in two levels of the PowerPC architecture. It also provides detailed descriptions of conventions used for storing values in registers and memory, accessing the core registers, and the representation of data in these registers. This chapter explains the following:

- Operand conventions
- e300 core instruction set

3.1 Operand Conventions

This section describes the integer and floating-point operand conventions. It also describes the big- and little-endian byte ordering for the e300 core. Note that floating-point instructions or operands are not supported on the e300c2 core.

3.1.1 Data Organization in Memory and Memory Operands

Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Memory operands may be bytes, half words, words, or double words, or, for the load/store multiple and move assist instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

3.1.2 Endian Modes and Byte Ordering

The PowerPC architecture supports both big- and little-endian byte ordering. The default byte and bit ordering is big-endian. See Section 3.1.2, “Byte Ordering,” in the *Programming Environments Manual*, for more information about big- and little-endian byte ordering. Note that the e300 core does not support the modified (PowerPC) little-endian mode present in previous PowerPC cores but supports true little-endian mode.

True little-endian mode is supported in the e300 core to minimize the impact on software porting from true little-endian systems. The true little-endian mode applies for all instruction fetches and data load and store operations to and from memory. The e300 powers up in one of two endian modes, big-endian mode or true little-endian mode, selected by the *tle* signal at the negation of $\overline{\textit{hreset}}$. The endian mode should be set at the negation of $\overline{\textit{hreset}}$, and should remain unchanged by software for the duration of the system operation.

Bit 4 of HID2, (HID2[LET]) is used in conjunction with MSR[LE] to indicate the endian mode of operation of the e300 core as shown in Table 3-1. Note that the e300 core no longer supports modified (PowerPC) little-endian mode as in previous PowerPC cores.

Table 3-1. Endian Mode Indication

MSR[LE]	HID2[LET]	Endian Mode
0	x	Big-endian
1	1	True little-endian

When the e300 core is in true little-endian mode, memory and I/O subsystems are treated as true little-endian. The following occurs when operating in true little-endian mode:

- The byte reversing for instruction occurs before the instruction is decoded.
- The byte reversing for data occurs when the data item is being moved to or from the GPR.

Therefore, the byte reversal in little-endian mode for load or store accesses occurs between memory or the data cache, and the register files for the e300 core.

3.1.3 Alignment and Misaligned Accesses

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the natural address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned. For a detailed discussion about memory operands, see Chapter 3, “Operand Conventions,” in the *Programming Environments Manual*.

Operands for single-register memory access instructions have the characteristics shown in Table 3-2. (Although not permitted as memory operands, quad words are shown because quad-word alignment is desirable for certain memory operands.)

Table 3-2. Memory Operands

Operand	Length	Addr[28–31] Aligned
Byte	8 bits	xxxx
Half word	2 bytes	xxx0
Word	4 bytes	xx00
Double word	8 bytes	x000
Quad word	16 bytes	0000

Note: An x in an address bit position indicates that the bit can be 0 or 1 regardless of the state of other address bits.

The concept of alignment is also applied more generally to data in memory. For example, a 12-byte data item is said to be word-aligned if its address is a multiple of four.

Implementation Notes—The following describes how the e300 core handles alignment and misaligned accesses:

- The core provides hardware support for some misaligned memory accesses. However, misaligned accesses suffer a performance degradation compared to aligned accesses of the same type.

- The core does not provide hardware support for floating-point load/store operations that are not word-aligned. In such a case, the core invokes an alignment interrupt and the interrupt handler must break up the misaligned access. For this reason, floating-point single- and double-word accesses should always be word-aligned. Note that a floating-point double-word access on a word-aligned boundary requires an extra cycle to complete. The e300c2 core does not support any floating-point operations.

Any half-word, word, double-word, and string reference access that crosses an alignment boundary must be broken into multiple discrete accesses. For string accesses, the hardware makes no attempt to get aligned to reduce the number of accesses. (Multiple word accesses are architecturally required to be aligned.) The resulting performance degradation depends on how well each individual access behaves with respect to the memory hierarchy. At a minimum, additional cache access cycles are required. More dramatically, each discrete access to a noncacheable page involves an individual bus operation that reduces the effective bus bandwidth.

The frequent use of misaligned accesses is discouraged because they can compromise the overall performance.

3.1.4 Floating-Point Execution Model

The e300c1 core provides hardware support for all single- and double-precision floating-point operations for most value representations and all rounding modes. The PowerPC architecture provides for hardware to implement a floating-point system as defined in ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating Point Arithmetic*. For detailed information about the floating-point execution model, refer to Chapter 3, “Operand Conventions,” in the *Programming Environments Manual*. Note that the e300c2 core does not support floating-point operations.

The IEEE 754 standard includes 64- and 32-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands.

The UISA follows these guidelines:

- Double-precision arithmetic instructions may have single-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision and always produce single-precision results.

For arithmetic instructions, conversions from double- to single-precision must be done explicitly by software, while conversions from single- to double-precision are done implicitly.

All PowerPC implementations provide the equivalent of the following execution models to ensure that identical results are obtained. The definition of the arithmetic instructions for infinities, denormalized numbers, and NaNs follow conventions described in the following sections.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit

computation of the adjusted exponent value in the following examples when the corresponding exception enable bit is one:

- Underflow during multiplication using a denormalized factor
- Overflow during division using a denormalized divisor

3.1.5 Effect of Operand Placement on Performance

The VEA states that the placement (location and alignment) of operands in memory affect the relative performance of memory accesses. The best performance is guaranteed if memory operands are aligned on natural boundaries. To obtain the best performance from the core, the programmer should assume the performance model described in Chapter 3, “Operand Conventions,” in the *Programming Environments Manual*.

3.2 Instruction Set Summary

This section describes instructions and addressing modes defined for the e300 core. These instructions are divided into the following functional categories:

- Integer instructions—These include arithmetic and logical instructions. For more information, see Section 3.2.4.1, “Integer Instructions.”
- Floating-point instructions—These include floating-point arithmetic instructions, as well as instructions that affect the floating-point status and control register (FPSCR). For more information, see Section 3.2.4.2, “Floating-Point Instructions.” Floating-point instructions are not supported on the e300c2 core.
- Load and store instructions—These include integer and floating-point load and store instructions. For more information, see Section 3.2.4.3, “Load and Store Instructions.”
- Flow control instructions—These include branching instructions, condition register logical instructions, and other instructions that affect the instruction flow. For more information, see Section 3.2.4.4, “Branch and Flow Control Instructions.”
- Trap instructions—These are used to test for a specified set of conditions; see Section 3.2.4.5, “Trap Instructions.”
- Processor control instructions—These are used for synchronizing memory accesses and managing caches, TLBs, and segment registers. For more information, see Section 3.2.4.6, “Processor Control Instructions,” Section 3.2.5.1, “Processor Control Instructions,” and Section 3.2.6.2, “Processor Control Instructions—OEA.”
- Memory synchronization instructions—These are used for synchronizing memory accesses. See Section 3.2.4.7, “Memory Synchronization Instructions—UISA,” and Section 3.2.5.2, “Memory Synchronization Instructions—VEA.”
- Memory control instructions—These provide control of caches, TLBs, and segment registers. For more information, see Section 3.2.5.3, “Memory Control Instructions—VEA,” and Section 3.2.6.3, “Memory Control Instructions—OEA.”
- System linkage instructions—These include the System Call (**sc**) and Return from Interrupt (**rfi**) instructions. See Section 3.2.6.1, “System Linkage Instructions.”

Note that this grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions. This information, which is useful in taking full advantage of the core superscalar parallel instruction execution, is provided in Chapter 8, “Instruction Set,” of the *Programming Environments Manual*.

Integer instructions operate on word operands. Floating-point instructions operate on single- and double-precision floating-point operands. PowerPC instructions are 4-byte words. The UISA provides for byte, half-word, and word operand loads and stores between memory and a set of 32 GPRs. It also provides for word and double-word operand loads and stores between memory and a set of 32 FPRs. Floating-point instructions and registers are not supported on the e300c2 core.

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

The description of each instruction includes the mnemonic and a formatted list of operands. To simplify assembly language programming, a set of simplified mnemonics (extended mnemonics in the architecture specification) and symbols is provided for some of the frequently-used instructions; see Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*, for a complete list of simplified mnemonic examples.

3.2.1 Classes of Instructions

The e300 core instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

Note that although the definitions of these terms are consistent among the processors of this family, the assignment of these classifications is not. For example, an instruction that is specific to 64-bit implementations is considered defined for 64-bit implementations but illegal for 32-bit implementations such as the e300 core.

The class is determined by examining the primary opcode and the extended opcode, if any. If either is not that of a defined instruction or of a reserved instruction, the instruction is illegal.

In future versions of the PowerPC architecture, instruction codings that are now illegal may become assigned to instructions in the architecture or may be reserved by being assigned to processor-specific instructions.

3.2.1.1 Definition of Boundedly Undefined

If instructions are encoded with incorrectly set bits in reserved fields, the results on execution can be said to be boundedly undefined. If a user-level program executes the incorrectly coded instruction, the resulting undefined results are bounded in that a spurious change from user to supervisor state is not allowed, and the level of privilege exercised by the program in relation to memory access and other system resources

cannot be exceeded. Boundedly undefined results for a given instruction may vary between implementations, and between execution attempts in the same implementation.

3.2.1.2 Defined Instruction Class

Defined instructions are guaranteed to be supported in all PowerPC implementations, except as stated in the instruction descriptions in Chapter 8, “Instruction Set,” in the *Programming Environments Manual*. The e300 core provides hardware support for all instructions defined for 32-bit implementations.

A processor of this family invokes the illegal instruction error handler (part of the program interrupt) when the unimplemented PowerPC instructions are encountered so they can be emulated in software, as required.

A defined instruction can have invalid forms, as described in the following section.

3.2.1.3 Illegal Instruction Class

Illegal instructions are grouped into the following categories:

- Instructions not defined in the PowerPC architecture. These opcodes are available for future extensions of the PowerPC architecture; that is, future versions of the PowerPC architecture may define any of these instructions to perform new functions.

  The following primary opcodes are defined as illegal but may be used in future extensions to the architecture:

  1, 4, 5, 6, 9, 22, 56, 57, 60, 61
- Instructions defined in the PowerPC architecture but not implemented in a specific PowerPC implementation. For example, instructions that can be executed on 64-bit processors are considered illegal by 32-bit processor cores.

  The following primary opcodes are defined for 64-bit implementations only and are illegal on the core:

  2, 30, 58, 62
- All unused extended opcodes are illegal. The unused extended opcodes can be determined from information in Appendix A.2, “Instructions Sorted by Opcode,” [and Section 3.2.1.4, “Reserved Instruction Class.”Notice that extended opcodes for instructions that are defined only for 64-bit implementations are illegal in 32-bit implementations, and vice versa.

  The following primary opcodes have unused extended opcodes:

  17, 19, 31, 59, 63 (primary opcodes 30 and 62 are illegal for all 32-bit implementations, but as 64-bit opcodes they have some unused extended opcodes)
- An instruction consisting entirely of zeros is guaranteed to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized memory invokes the system illegal instruction error handler (a program interrupt). Note that if only the primary opcode consists of all zeros, the instruction is considered a reserved instruction. This is further described in Section 3.2.1.4, “Reserved Instruction Class.”

An attempt to execute an illegal instruction invokes the illegal instruction error handler (a program interrupt) but has no other effect. Section 5.5.7, “Program Interrupt (0x00700),” describes illegal and invalid instruction interrupts.

Except for an instruction consisting entirely of binary zeros, illegal instructions are available for further additions to the PowerPC architecture.

3.2.1.4 Reserved Instruction Class

Reserved instructions are allocated to specific implementation-dependent purposes not defined by the PowerPC architecture. An attempt to execute an unimplemented reserved instruction invokes the illegal instruction error handler (a program interrupt). See Section 5.5.7, “Program Interrupt (0x00700),” for additional information about illegal and invalid instruction interrupts.

The following types of instructions are included in this class:

- Implementation-specific instructions (for example, Load Data TLB Entry (**tlbld**) and Load Instruction TLB Entry (**tlbli**) instructions).
- Optional instructions defined by the PowerPC architecture but not implemented by the core (for example, Floating Square Root (**fsqrt**) and Floating Square Root Single (**fsqrts**) instructions).

3.2.2 Addressing Modes

This section provides an overview of conventions for addressing memory and calculating effective addresses as defined by the PowerPC architecture for 32-bit implementations. For more detailed information, see “Conventions” in Chapter 4, “Addressing Modes and Instruction Set Summary,” of the *Programming Environments Manual*.

3.2.2.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a memory access or branch instruction or when it fetches the next sequential instruction.

As described in Section 3.1.1, “Data Organization in Memory and Memory Operands,” bytes in memory are numbered consecutively starting with zero. Each number is the address of the corresponding byte.

3.2.2.2 Memory Operands

Memory operands may be bytes, half words, words, or double words, or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction. The PowerPC architecture supports both big- and little-endian byte ordering. The default byte and bit ordering is big-endian. See Section 3.1.2, “Byte Ordering,” in the *Programming Environments Manual*, for more information about big- and little-endian byte ordering.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the “natural” address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is

misaligned. For a detailed discussion about memory operands, see Chapter 3, “Operand Conventions,” in the *Programming Environments Manual*.

3.2.2.3 Effective Address Calculation

An effective address (EA) is the 32-bit sum computed by the processor core when executing a memory access or branch instruction or when fetching the next sequential instruction. For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the memory operand is considered to wrap around from the maximum effective address through effective address 0, as described in the following paragraphs.

Effective address computations for both data and instruction accesses use 32-bit unsigned binary arithmetic. A carry from bit 0 is ignored.

Load and store operations have three categories of effective address generation:

- Register indirect with immediate index mode
- Register indirect with index mode
- Register indirect mode

Section 3.2.4.3.2, “Integer Load and Store Address Generation,” describes effective address generation for load and store operations.

Branch instructions have three categories of effective address generation:

- Immediate
- Link register indirect
- Count register indirect

Section 3.2.4.4.1, “Branch Instruction Address Calculation,” describes branch instruction effective address generation.

3.2.2.4 Synchronization

The synchronization described in this section refers to the state of the core performing the synchronization.

3.2.2.4.1 Context Synchronization

The System Call (**sc**) and Return from Interrupt (**rfi**) instructions perform context synchronization by allowing previously issued instructions to complete before performing a change in context. Execution of one of these instructions ensures the following:

- No higher priority interrupt exists (**sc**).
- All previous instructions have completed to a point where they can no longer cause an interrupt. If a prior memory access instruction causes direct-store error interrupts, the results are guaranteed to be determined before this instruction is executed.

- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The instructions following the **sc** or **rfi** instruction execute in the context established by these instructions.

3.2.2.4.2 Execution Synchronization

An instruction is execution synchronizing if all previously initiated instructions appear to have completed before the instruction is initiated or, in the case of the Synchronize (**sync**) and Instruction Synchronize (**isync**) instructions, before the instruction completes. For example, the Move to Machine State Register (**mtmsr**) instruction is execution synchronizing. It ensures that all preceding instructions have completed execution and will not cause an interrupt before the instruction executes but does not ensure subsequent instructions execute in the newly established environment. For example, if the **mtmsr** sets MSR[PR], unless an **isync** immediately follows the **mtmsr** instruction, a privileged instruction could be executed or privileged access could be performed without causing an interrupt even though MSR[PR] indicates user mode.

3.2.2.4.3 Instruction-Related Interrupts

There are two kinds of interrupts in the e300 core—those caused directly by the execution of an instruction and those caused by an asynchronous event. Either may cause components of the system software to be invoked.

Interrupts can be caused directly by the execution of an instruction as follows:

- An attempt to execute an illegal instruction causes the illegal instruction (program interrupt) handler to be invoked. An attempt by a user-level program to execute the supervisor-level instructions listed below causes the privileged instruction (program interrupt) handler to be invoked. The core provides the following supervisor-level instructions: **icbt, dcbi**, **mfmsr**, **mfspr**, **mfsr**, **mfsrin**, **mtmsr**, **mtspr**, **mtsr**, **mtsrin**, **rfi**, **tlbie**, **tlbsync**, **tlbld**, and **tlbli**. Note that the privilege level of the **mfspr** and **mtspr** instructions depends on the SPR encoding.
- An attempt to access memory that is not available (page fault) causes the ISI interrupt handler to be invoked.
- An attempt to access memory with an effective address alignment that is invalid for the instruction causes the alignment interrupt handler to be invoked.
- The execution of an **sc** instruction invokes the system call interrupt handler that permits a program to request the system to perform a service.
- The execution of a trap instruction invokes the program interrupt trap handler.
- The execution of a floating-point instruction when floating-point instructions are disabled or unavailable invokes the floating-point unavailable interrupt handler.
- The execution of an instruction that causes a floating-point exception while exceptions are enabled in the MSR invokes the program interrupt handler. Floating-point instructions are not supported on the e300c2 core.

Interrupts caused by asynchronous events are described in Chapter 5, “Interrupts and Exceptions.”

3.2.3 Instruction Set Overview

This section provides a brief overview of the PowerPC instructions implemented in the core and highlights any special information with respect to how the e300 core implements a particular instruction. Note that the categories used in this section correspond to those used in Chapter 4, “Addressing Modes and Instruction Set Summary,” in the *Programming Environments Manual*. These categorizations are somewhat arbitrary and are provided for the convenience of the programmer and do not necessarily reflect the PowerPC architecture specification.

Note that some of the instructions have the following optional features:

- CR Update—The dot (**.**) suffix on the mnemonic enables the update of the CR.
- Overflow option—The **o** suffix indicates that the overflow bit in the XER is enabled.

3.2.4 PowerPC UISA Instructions

The UISA includes the base user-level instruction set (excluding a few user-level cache control, synchronization, and time base instructions), user-level registers, programming model, data types, and addressing modes. This section discusses the instructions defined in the UISA.

3.2.4.1 Integer Instructions

This section describes the integer instructions. These consist of the following:

- Integer arithmetic instructions
- Integer compare instructions
- Integer logical instructions
- Integer rotate and shift instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs, into the XER, and into condition register (CR) fields.

3.2.4.1.1 Integer Arithmetic Instructions

Table 3-3 lists the integer arithmetic instructions for the core.

Table 3-3. Integer Arithmetic Instructions

Name	Mnemonic	Operand Syntax
Add	**add** (**add.** **addo** **addo.**)	**r**D,**r**A,**r**B
Add Carrying	**addc** (**addc.** **addco** **addco.**)	**r**D,**r**A,**r**B
Add Extended	**adde** (**adde.** **addeo** **addeo.**)	**r**D,**r**A,**r**B
Add Immediate	**addi**	**r**D,**r**A,SIMM
Add Immediate Carrying	**addic**	**r**D,**r**A,SIMM
Add Immediate Carrying and Record	**addic.**	**r**D,**r**A,SIMM
Add Immediate Shifted	**addis**	**r**D,**r**A,SIMM
Add to Minus One Extended	**addme** (**addme.** **addmeo** **addmeo.**)	**r**D,**r**A

Table 3-3. Integer Arithmetic Instructions (continued)

Name	Mnemonic	Operand Syntax
Add to Zero Extended	**addze** (**addze. addzeo addzeo.**)	**r**D,**r**A
Divide Word	**divw** (**divw. divwo divwo.**)	**r**D,**r**A,**r**B
Divide Word Unsigned	**divwu** (**divwu. divwuo divwuo.**)	**r**D,**r**A,**r**B
Multiply High Word	**mulhw** (**mulhw.**)	**r**D,**r**A,**r**B
Multiply High Word Unsigned	**mulhwu** (**mulhwu.**)	**r**D,**r**A,**r**B
Multiply Low	**mullw** (**mullw. mullwo mullwo.**)	**r**D,**r**A,**r**B
Multiply Low Immediate	**mulli**	**r**D,**r**A,SIMM
Negate	**neg** (**neg. nego nego.**)	**r**D,**r**A
Subtract From	**subf** (**subf. subfo subfo.**)	**r**D,**r**A,**r**B
Subtract From Carrying	**subfc** (**subfc. subfco subfco.**)	**r**D,**r**A,**r**B
Subtract From Extended	**subfe** (**subfe. subfeo subfeo.**)	**r**D,**r**A,**r**B
Subtract From Immediate Carrying	**subfic**	**r**D,**r**A,SIMM
Subtract From Minus One Extended	**subfme** (**subfme. subfmeo subfmeo.**)	**r**D,**r**A
Subtract From Zero Extended	**subfze** (**subfze. subfzeo subfzeo.**)	**r**D,**r**A

Although there is no Subtract Immediate instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. The **subf** instructions subtract the second operand (**r**A) from the third operand (**r**B). Simplified mnemonics are provided in which the third operand is subtracted from the second operand. See Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*, for examples.

3.2.4.1.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of **r**A with either the UIMM operand, the SIMM operand, or the contents of **r**B. The comparison is signed for the **cmpi** and **cmp** instructions, and unsigned for the **cmpli** and **cmpl** instructions. Table 3-4 lists the integer compare instructions.

Table 3-4. Integer Compare Instructions

Name	Mnemonic	Operand Syntax
Compare	**cmp**	**crf**D,L,**r**A,**r**B
Compare Immediate	**cmpi**	**crf**D,L,**r**A,SIMM
Compare Logical	**cmpl**	**crf**D,L,**r**A,**r**B
Compare Logical Immediate	**cmpli**	**crf**D,L,**r**A,UIMM

The **crf**D operand can be omitted if the result of the comparison is to be placed in CR0. Otherwise, the target CR field must be specified in the instruction **crf**D field.

For more information refer to Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*.

3.2.4.1.3 Integer Logical Instructions

The logical instructions shown in Table 3-5 perform bit-parallel operations. Logical instructions with the CR update enabled and instructions **andi.** and **andis.** set CR field CR0 to characterize the result of the logical operation. These fields are set as if the sign-extended low-order 32 bits of the result were algebraically compared to zero. Logical instructions without CR update and the remaining logical instructions do not modify the CR. Logical instructions do not affect the XER[SO], XER[OV], and XER[CA] bits.

For simplified mnemonics examples for the integer logical operations see Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*.

Table 3-5. Integer Logical Instructions

Name	Mnemonic	Operand Syntax
AND	**and** (**and.**)	**r**A,**r**S,**r**B
AND Immediate	**andi.**	**r**A,**r**S,UIMM
AND Immediate Shifted	**andis.**	**r**A,**r**S,UIMM
AND with Complement	**andc** (**andc.**)	**r**A,**r**S,**r**B
Count Leading Zeros Word	**cntlzw** (**cntlzw.**)	**r**A,**r**S
Equivalent	**eqv** (**eqv.**)	**r**A,**r**S,**r**B
Extend Sign Byte	**extsb** (**extsb.**)	**r**A,**r**S
Extend Sign Half Word	**extsh** (**extsh.**)	**r**A,**r**S
NAND	**nand** (**nand.**)	**r**A,**r**S,**r**B
NOR	**nor** (**nor.**)	**r**A,**r**S,**r**B
OR	**or** (**or.**)	**r**A,**r**S,**r**B
OR Immediate	**ori**	**r**A,**r**S,UIMM
OR Immediate Shifted	**oris**	**r**A,**r**S,UIMM
OR with Complement	**orc** (**orc.**)	**r**A,**r**S,**r**B
XOR	**xor** (**xor.**)	**r**A,**r**S,**r**B
XOR Immediate	**xori**	**r**A,**r**S,UIMM
XOR Immediate Shifted	**xoris**	**r**A,**r**S,UIMM

3.2.4.1.4 Integer Rotate and Shift Instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. See Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*, for a complete list of simplified mnemonics that allows simpler coding of often-used functions such as clearing the left-most or right-most bits of a register, left justifying or right justifying an arbitrary field, and simple rotates and shifts.

Integer rotate instructions rotate the contents of a register. The result of the rotation is either inserted into the target register under control of a mask (if a mask bit is 1, the associated bit of the rotated data is placed into the target register; and if the mask bit is 0, the associated bit in the target register is unchanged), or ANDed with a mask before being placed into the target register.

The integer rotate instructions are listed in Table 3-6.

Table 3-6. Integer Rotate Instructions

Name	Mnemonic	Operand Syntax
Rotate Left Word Immediate then AND with Mask	**rlwinm** (**rlwinm.**)	**r**A,**r**S,SH,MB,ME
Rotate Left Word Immediate then Mask Insert	**rlwimi** (**rlwimi.**)	**r**A,**r**S,SH,MB,ME
Rotate Left Word then AND with Mask	**rlwnm** (**rlwnm.**)	**r**A,**r**S,**r**B,MB,ME

The integer shift instructions perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics are provided, making coding of such shifts simpler and easier to understand.

Multiple-precision shifts can be programmed as shown in Appendix C, “Multiple-Precision Shifts,” in the *Programming Environments Manual*.

The integer shift instructions are listed in Table 3-7.

Table 3-7. Integer Shift Instructions

Name	Mnemonic	Operand Syntax
Shift Left Word	**slw** (**slw.**)	**r**A,**r**S,**r**B
Shift Right Algebraic Word	**sraw** (**sraw.**)	**r**A,**r**S,**r**B
Shift Right Algebraic Word Immediate	**srawi** (**srawi.**)	**r**A,**r**S,SH
Shift Right Word	**srw** (**srw.**)	**r**A,**r**S,**r**B

3.2.4.2 Floating-Point Instructions

This section describes the floating-point instructions, which include the following:

- Floating-point arithmetic instructions
- Floating-point multiply-add instructions
- Floating-point rounding and conversion instructions
- Floating-point compare instructions
- Floating-point status and control register instructions
- Floating-point move instructions

See Section 3.2.4.3, “Load and Store Instructions,” for information about floating-point loads and stores.

Implementation Note—The e300c2 core does not support floating-point instructions.

The PowerPC architecture supports a floating-point system as defined in the IEEE 754 standard, but requires software support to conform with that standard. All floating-point operations conform to the IEEE 754 standard, except if software sets the non-IEEE mode bit (NI) in the FPSCR. The e300 core is in the

nondenormalized mode when the NI bit is set in the FPSCR. If a denormalized result is produced, a default result of zero is generated. The generated zero has the same sign as the denormalized number. The core performs single- and double-precision floating-point operations compliant with the IEEE 754 floating-point standard.

Implementation note—Single-precision denormalized results require two additional processor clock cycles to round. When loading or storing a single-precision denormalized number, the load/store unit may take up to 24 processor clock cycles to convert between the internal double-precision format and the external single-precision format.

3.2.4.2.1 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are listed in Table 3-8.

Table 3-8. Floating-Point Arithmetic Instructions

Name	Mnemonic	Operand Syntax
Floating Add (Double-Precision)	**fadd** (**fadd.**)	**fr**D,**fr**A,**fr**B
Floating Add Single	**fadds** (**fadds.**)	**fr**D,**fr**A,**fr**B
Floating Divide (Double-Precision)	**fdiv** (**fdiv.**)	**fr**D,**fr**A,**fr**B
Floating Divide Single	**fdivs** (**fdivs.**)	**fr**D,**fr**A,**fr**B
Floating Multiply (Double-Precision)	**fmul** (**fmul.**)	**fr**D,**fr**A,**fr**C
Floating Multiply Single	**fmuls** (**fmuls.**)	**fr**D,**fr**A,**fr**C
Floating Reciprocal Estimate Single	**fres** (**fres.**)	**fr**D,**fr**B
Floating Reciprocal Square Root Estimate	**frsqrte** (**frsqrte.**)	**fr**D,**fr**B
Floating Select	**fsel** (**fsel.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Subtract (Double-Precision)	**fsub** (**fsub.**)	**fr**D,**fr**A,**fr**B
Floating Subtract Single	**fsubs** (**fsubs.**)	**fr**D,**fr**A,**fr**B

3.2.4.2.2 Floating-Point Multiply-Add Instructions

These instructions combine multiply and add operations without an intermediate rounding operation. The fractional part of the intermediate product is 106 bits wide, and all 106 bits take part in the add/subtract portion of the instruction.

The floating-point multiply-add instructions are listed in Table 3-9.

Table 3-9. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Operand Syntax
Floating Multiply-Add (Double-Precision)	**fmadd** (**fmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Add Single	**fmadds** (**fmadds.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Subtract (Double-Precision)	**fmsub** (**fmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Subtract Single	**fmsubs** (**fmsubs.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Add (Double-Precision)	**fnmadd** (**fnmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B

Table 3-9. Floating-Point Multiply-Add Instructions (continued)

Name	Mnemonic	Operand Syntax
Floating Negative Multiply-Add Single	**fnmadds** (**fnmadds.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Subtract (Double-Precision)	**fnmsub** (**fnmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Subtract Single	**fnmsubs** (**fnmsubs.**)	**fr**D,**fr**A,**fr**C,**fr**B

Implementation note—Single-precision multiply-type instructions operate faster than their double-precision equivalents. See Chapter 7, “Instruction Timing,” for more information.

3.2.4.2.3 Floating-Point Rounding and Conversion Instructions

The Floating Round to Single-Precision (**frsp**) instruction is used to truncate a 64-bit double-precision number to a 32-bit single-precision floating-point number. The floating-point conversion instructions convert a 64-bit double-precision floating-point number to a 32-bit signed integer number.

The PowerPC architecture defines bits 0–31 of floating-point register **fr**D as undefined when executing the Floating Convert to Integer Word (**fctiw**) and Floating Convert to Integer Word with Round Toward Zero (**fctiwz**) instructions.

Examples of uses of these instructions to perform various conversions can be found in Appendix D, “Floating-Point Models,” in the *Programming Environments Manual*. The floating-point rounding instructions are shown in Table 3-10.

Table 3-10. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Operand Syntax
Floating Convert to Integer Word	**fctiw** (**fctiw.**)	**fr**D,**fr**B
Floating Convert to Integer Word with Round Toward Zero	**fctiwz** (**fctiwz.**)	**fr**D,**fr**B
Floating Round to Single-Precision	**frsp** (**frsp.**)	**fr**D,**fr**B

3.2.4.2.4 Floating-Point Compare Instructions

Floating-point compare instructions compare the contents of two floating-point registers. The comparison ignores the sign of zero (that is +0 = –0). The floating-point compare instructions are listed in Table 3-11.

Table 3-11. Floating-Point Compare Instructions

Name	Mnemonic	Operand Syntax
Floating Compare Ordered	**fcmpo**	**crf**D,**fr**A,**fr**B
Floating Compare Unordered	**fcmpu**	**crf**D,**fr**A,**fr**B

3.2.4.2.5 Floating-Point Status and Control Register Instructions

Every FPSCR instruction appears to synchronize the effects of all floating-point instructions executed by a given processor. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the given processor appear to have completed before the FPSCR instruction is initiated and

that no subsequent floating-point instructions appear to be initiated by the given processor until the FPSCR instruction has completed. The FPSCR instructions are listed in Table 3-12.

Table 3-12. Floating-Point Status and Control Register Instructions

Name	Mnemonic	Operand Syntax
Move from FPSCR	**mffs** (**mffs.**)	**fr**D
Move to Condition Register from FPSCR	**mcrfs**	**crf**D,**crf**S
Move to FPSCR Bit 0	**mtfsb0** (**mtfsb0.**)	**crb**D
Move to FPSCR Bit 1	**mtfsb1** (**mtfsb1.**)	**crb**D
Move to FPSCR Field Immediate	**mtfsfi** (**mtfsfi.**)	**crf**D,IMM
Move to FPSCR Fields	**mtfsf** (**mtfsf.**)	FM,**fr**B

Implementation Note—The architecture notes that, in some implementations, the Move to FPSCR Fields (**mtfsf***x*) instruction may perform more slowly when only a portion of the fields are updated as opposed to all of the fields. This is not the case in the e300 core.

3.2.4.2.6 Floating-Point Move Instructions

Floating-point move instructions copy data from one floating-point register to another. The floating-point move instructions do not modify the FPSCR. The CR update option in these instructions controls the placing of result status into CR1. Floating-point move instructions are listed in Table 3-13.

Table 3-13. Floating-Point Move Instructions

Name	Mnemonic	Operand Syntax
Floating Absolute Value	**fabs** (**fabs.**)	**fr**D,**fr**B
Floating Move Register	**fmr** (**fmr.**)	**fr**D,**fr**B
Floating Negate	**fneg** (**fneg.**)	**fr**D,**fr**B
Floating Negative Absolute Value	**fnabs** (**fnabs.**)	**fr**D,**fr**B

3.2.4.3 Load and Store Instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering. This section describes the load and store instructions of the e300 core, which consist of the following:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Integer load and store string instructions
- Floating-point load instructions
- Floating-point store instructions

3.2.4.3.1 Self-Modifying Code

When a processor modifies a memory location that may be contained in the instruction cache, software must ensure that memory updates are visible to the instruction fetching mechanism. This can be achieved by the following instruction sequence:

```
dcbst    |update memory
sync     |wait for update
icbi     |remove (invalidate) copy in instruction cache
isync    |remove copy in own instruction buffer
```

These operations are required because the data cache is a write-back cache. Since instruction fetching bypasses the data cache, changes to items in the data cache may not be reflected in memory until the fetch operations complete.

Special care must be taken to avoid coherency paradoxes in systems that implement unified secondary caches, and designers should carefully follow the guidelines for maintaining cache coherency that are provided in the VEA, and discussed in Chapter 5, “Cache Model and Memory Coherency,” in the *Programming Environments Manual*. Because the core does not broadcast the M bit for instruction fetches (except when HID0[IFEM] is set), external caches are subject to coherency paradoxes.

3.2.4.3.2 Integer Load and Store Address Generation

Integer load and store operations generate effective addresses using register indirect with immediate index mode, register indirect with index mode, or register indirect mode. See Section 3.2.2.3, “Effective Address Calculation.” Note that the core is optimized for load and store operations that are aligned on natural boundaries, and operations that are not naturally aligned may suffer performance degradation. Refer to Section 5.5.6.1, “Integer Alignment Exceptions.”

3.2.4.3.3 Register Indirect Integer Load Instructions

For integer load instructions, the byte, half word, word, or double word addressed by the EA is loaded into **r**D. Many integer load instructions have an update form, in which **r**A is updated with the generated effective address. For these forms, the EA is placed into **r**A and the memory element (byte, half word, word, or double word) addressed by EA is loaded into **r**D.

Implementation Note—In some implementations, the load half word algebraic instructions (**lha** and **lhax**) and the load with update (**lbzu**, **lbzux**, **lhzu**, **lhzux**, **lhau**, **lhaux**, **lwu**, and **lwux**) instructions may execute with greater latency than other types of load instructions. In the e300 core, these instructions operate with the same latency as other load instructions.

Table 3-14 lists the integer load instructions.

Table 3-14. Integer Load Instructions

Name	Mnemonic	Operand Syntax
Load Byte and Zero	**lbz**	**r**D,d(**r**A)
Load Byte and Zero Indexed	**lbzx**	**r**D,**r**A,**r**B
Load Byte and Zero with Update	**lbzu**	**r**D,d(**r**A)
Load Byte and Zero with Update Indexed	**lbzux**	**r**D,**r**A,**r**B

Table 3-14. Integer Load Instructions (continued)

Name	Mnemonic	Operand Syntax
Load Half Word Algebraic	**lha**	**r**D,d(**r**A)
Load Half Word Algebraic Indexed	**lhax**	**r**D,**r**A,**r**B
Load Half Word Algebraic with Update	**lhau**	**r**D,d(**r**A)
Load Half Word Algebraic with Update Indexed	**lhaux**	**r**D,**r**A,**r**B
Load Half Word and Zero	**lhz**	**r**D,d(**r**A)
Load Half Word and Zero Indexed	**lhzx**	**r**D,**r**A,**r**B
Load Half Word and Zero with Update	**lhzu**	**r**D,d(**r**A)
Load Half Word and Zero with Update Indexed	**lhzux**	**r**D,**r**A,**r**B
Load Word and Zero	**lwz**	**r**D,d(**r**A)
Load Word and Zero Indexed	**lwzx**	**r**D,**r**A,**r**B
Load Word and Zero with Update	**lwzu**	**r**D,d(**r**A)
Load Word and Zero with Update Indexed	**lwzux**	**r**D,**r**A,**r**B

3.2.4.3.4 Integer Store Instructions

For integer store instructions, the contents of **r**S are stored into the byte, half word, word, or double word in memory addressed by the effective address. Many store instructions have an update form, in which **r**A is updated with the EA. For these forms, the following rules apply:

- If **r**A ≠ 0, the EA is placed into **r**A.
- If **r**S = **r**A, the contents of **r**S are copied to the target memory element, then the generated EA is placed into **r**A (**r**S).

The core defines store with update instructions with **r**A = 0 and integer store instructions with the CR update option enabled (Rc field, bit 31, in the instruction encoding = 1) to be invalid forms. Table 3-15 provides a list of the integer store instructions for the core.

Table 3-15. Integer Store Instructions

Name	Mnemonic	Operand Syntax
Store Byte	**stb**	**r**S,d(**r**A)
Store Byte Indexed	**stbx**	**r**S,**r**A,**r**B
Store Byte with Update	**stbu**	**r**S,d(**r**A)
Store Byte with Update Indexed	**stbux**	**r**S,**r**A,**r**B
Store Half Word	**sth**	**r**S,d(**r**A)
Store Half Word Indexed	**sthx**	**r**S,**r**A,**r**B
Store Half Word with Update	**sthu**	**r**S,d(**r**A)
Store Half Word with Update Indexed	**sthux**	**r**S,**r**A,**r**B
Store Word	**stw**	**r**S,d(**r**A)
Store Word Indexed	**stwx**	**r**S,**r**A,**r**B

Table 3-15. Integer Store Instructions (continued)

Name	Mnemonic	Operand Syntax
Store Word with Update	**stwu**	**r**S,d(**r**A)
Store Word with Update Indexed	**stwux**	**r**S,**r**A,**r**B

3.2.4.3.5 Integer Load and Store with Byte-Reverse Instructions

Table 3-16 describes integer load and store with byte-reverse instructions. When used in a system operating with the default big-endian byte order, these instructions have the effect of loading and storing data in little-endian order. Likewise, when used in a system operating with little-endian byte order, these instructions have the effect of loading and storing data in big-endian order. When operating with true little-endian byte order, these instructions have the effect of loading and storing data in true little-endian order. For more information about big- and little-endian byte ordering, see Section 3.1.2, “Byte Ordering,” in the *Programming Environments Manual*. For more information about true little-endian operation, see Section 3.1.2, “Endian Modes and Byte Ordering.”

The e300 core supports the true little-endian mode. In true little-endian mode, the core treats the memory and I/O subsystems as little-endian memory. In this case, instruction and data bytes are reserved as follows:

- The byte reversing for instruction accesses occurs before the instruction is decoded.
- The byte reversing occurs for data accesses when the data item is being moved to or from the GPR.

Therefore, byte reversal during the load or store accesses is performed between memory or the data cache, and the register files.

Table 3-16. Integer Load and Store with Byte-Reverse Instructions

Name	Mnemonic	Operand Syntax
Load Half Word Byte-Reverse Indexed	**lhbrx**	**r**D,**r**A,**r**B
Load Word Byte-Reverse Indexed	**lwbrx**	**r**D,**r**A,**r**B
Store Half Word Byte-Reverse Indexed	**sthbrx**	**r**S,**r**A,**r**B
Store Word Byte-Reverse Indexed	**stwbrx**	**r**S,**r**A,**r**B

Implementation Note—In some implementations, load byte-reverse instructions (**lhbrx** and **lwbrx**) may have greater latency than other load instructions; however, these instructions operate with the same latency as other load instructions in the core.

3.2.4.3.6 Integer Load and Store Multiple Instructions

The integer load/store multiple instructions are used to move blocks of data to and from the GPRs. In some implementations, these instructions are likely to have greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Implementation notes—The following describes the e300 core implementation of the load/store multiple instruction:

- The load multiple and store multiple instructions may have operands that require memory accesses crossing a 4-Kbyte page boundary. As a result, these instructions may be interrupted by a DSI interrupt associated with the address translation of the second page. In this case, the core performs some or all of the memory references from the first page, and none of the memory references from the second page before taking the interrupt. On return from the DSI interrupt, the load or store multiple instruction will re-execute from the beginning. For additional information, refer to "DSI Interrupt (0x00300)" in Chapter 6, "Interrupts," in the *Programming Environments Manual*.
- The PowerPC architecture defines the load multiple word (**lmw**) instruction with **r**A in the range of registers to be loaded as an invalid form. It defines the load multiple and store multiple instructions with misaligned operands (that is, the EA is not a multiple of four) to cause an alignment interrupt. The core defines the load multiple word (**lmw**) instruction with **r**A in the range of registers to be loaded as an invalid form.
- The PowerPC architecture describes some preferred instruction forms for the integer load and store multiple instructions that may perform better than other forms in some implementations. None of these preferred forms affect instruction performance in the core.

Table 3-17. Integer Load and Store Multiple Instructions

Name	Mnemonic	Operand Syntax
Load Multiple Word	**lmw**	**r**D,d(**r**A)
Store Multiple Word	**stmw**	**r**S,d(**r**A)

3.2.4.3.7 Integer Load and Store String Instructions

The integer load and store string instructions allow movement of data from memory to registers or from registers to memory without concern for alignment. These instructions can be used for a short move between arbitrary memory locations or to initiate a long move between misaligned memory fields.

When the core is operating with little-endian byte order, execution of a load or store string instruction causes the system alignment error handler to be invoked; see Section 3.1.2, "Byte Ordering," in the *Programming Environments Manual*, for more information.

Table 3-18 lists the integer load and store string instructions.

Table 3-18. Integer Load and Store String Instructions

Name	Mnemonic	Operand Syntax
Load String Word Immediate	**lswi**	**r**D,**r**A,NB
Load String Word Indexed	**lswx**	**r**D,**r**A,**r**B
Store String Word Immediate	**stswi**	**r**S,**r**A,NB
Store String Word Indexed	**stswx**	**r**S,**r**A,**r**B

Load string and store string instructions may involve operands that are not word-aligned. As described in "Alignment Interrupt (0x00600)" in Chapter 6, "Interrupts," in the *Programming Environments Manual*,

a misaligned string operation suffers a performance penalty compared to a word-aligned operation of the same type.

When a string operation crosses a 4-Kbyte boundary, the instruction may be interrupted by a DSI interrupt associated with the address translation of the second page. In this case, the core performs some or all memory references from the first page and none from the second before taking the interrupt. On return from the DSI interrupt, the load or store string instruction will re-execute from the beginning. For more information, refer to “DSI Interrupt (0x00300)” in Chapter 6, “Interrupts,” in the *Programming Environments Manual*.

Implementation Note—If **r**A is in the range of registers to be loaded for a Load String Word Immediate (**lswi**) instruction or if either **r**A or **r**B is in the range of registers to be loaded for a Load String Word Indexed (**lswx**) instruction, the PowerPC architecture defines the instruction to be of an invalid form. In addition, the **lswx** and **stswx** instructions that specify a string length of zero are defined to be invalid by the PowerPC architecture. However, none of these cases hold true for the e300 core—the core treats these cases as valid forms.

3.2.4.3.8 Floating-Point Load and Store Address Generation

Floating-point load and store operations generate effective addresses using the register indirect with immediate index addressing mode and register indirect with index addressing mode (details are described below). Floating-point loads and stores are not supported for direct-store accesses. The use of the floating-point load and store operations for direct-store accesses results in a DSI interrupt.

Implementation Note—The e300c2 core does not support floating-point operations.

3.2.4.3.9 Floating-Point Load Instructions

Separate floating-point load instructions are used for single-precision and double-precision operands. Because FPRs support only double-precision format, the FPU converts single-precision data to double-precision format before loading the operands into the target FPR. This conversion is described fully in “Floating-Point Load Instructions” in Appendix D, “Floating-Point Models,” in the *Programming Environments Manual*.

Implementation Note—The PowerPC architecture defines load with update instructions with **r**A = 0 as an invalid form; however, the core treats this case as a valid form.

Table 3-19 provides a list of the floating-point load instructions.

Table 3-19. Floating-Point Load Instructions

Name	Mnemonic	Operand Syntax
Load Floating-Point Double	**lfd**	**fr**D,d(**r**A)
Load Floating-Point Double Indexed	**lfdx**	**fr**D,**r**A,**r**B
Load Floating-Point Double with Update	**lfdu**	**fr**D,d(**r**A)
Load Floating-Point Double with Update Indexed	**lfdux**	**fr**D,**r**A,**r**B
Load Floating-Point Single	**lfs**	**fr**D,d(**r**A)
Load Floating-Point Single Indexed	**lfsx**	**fr**D,**r**A,**r**B

Table 3-19. Floating-Point Load Instructions (continued)

Name	Mnemonic	Operand Syntax
Load Floating-Point Single with Update	**lfsu**	**fr**D,d(**r**A)
Load Floating-Point Single with Update Indexed	**lfsux**	**fr**D,**r**A,**r**B

3.2.4.3.10 Floating-Point Store Instructions

There are three basic forms of the store instruction—single-precision, double-precision, and integer. The integer form is supported by the optional **stfiwx** instruction. Because the FPRs support only double-precision format for floating-point data, the FPU converts double-precision data to single-precision format before storing the operands. The conversion steps are described in "Floating-Point Store Instructions" in Appendix D, "Floating-Point Models," in the *Programming Environments Manual*.

Implementation Note—The PowerPC architecture defines store with update instructions with **r**A = 0 as an invalid form; however, the core treats this case as valid.

Table 3-20 lists the floating-point store instructions.

Table 3-20. Floating-Point Store Instructions

Name	Mnemonic	Operand Syntax
Store Floating-Point as Integer Word Indexed	**stfiwx**	**fr**S,**r**A,**r**B
Store Floating-Point Double	**stfd**	**fr**S,d(**r**A)
Store Floating-Point Double Indexed	**stfdx**	**fr**S,**r**A,**r**B
Store Floating-Point Double with Update	**stfdu**	**fr**S,d(**r**A)
Store Floating-Point Double with Update Indexed	**stfdux**	**fr**S,**r**A,**r**B
Store Floating-Point Single	**stfs**	**fr**S,d(**r**A)
Store Floating-Point Single Indexed	**stfsx**	**fr**S,**r**A,**r**B
Store Floating-Point Single with Update	**stfsu**	**fr**S,d(**r**A)
Store Floating-Point Single with Update Indexed	**stfsux**	**fr**S,**r**A,**r**B

3.2.4.4 Branch and Flow Control Instructions

Branch instructions are executed by the branch processing unit (BPU). The BPU receives branch instructions from the fetch unit and performs CR lookahead operations on conditional branches to resolve them early, achieving the effect of a zero-cycle branch in many cases.

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. When the branch processor encounters one of these instructions, it scans the execution pipelines to determine whether an instruction in progress may affect the particular CR bit. If no interlock is found, the branch can be resolved immediately by checking the bit in the CR and taking the action defined for the branch instruction.

If an interlock is detected, the branch is considered unresolved and the direction of the branch is predicted using static branch prediction as described in "Conditional Branch Control" in Chapter 4, "Addressing Modes and Instruction Set Summary," in the *Programming Environments Manual*. The interlock is

monitored while instructions are fetched for the predicted branch. When the interlock is cleared, the branch processor determines whether the prediction was correct, based on the value of the CR bit. If the prediction is correct, the branch is considered completed and instruction fetching continues. If the prediction is incorrect, the fetched instructions are purged, and instruction fetching continues along the alternate path. See Chapter 8, “Instruction Timing,” in the *Programming Environments Manual*, for more information about how branches are executed.

3.2.4.4.1 Branch Instruction Address Calculation

Branch instructions can change the instruction sequence. Instruction addresses are always assumed to be word aligned; the processor ignores the two low-order bits of the generated branch target address.

Branch instructions compute the effective address (EA) of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register
- Branch conditional to count register

3.2.4.4.2 Branch Instructions

Table 3-21 lists the branch instructions provided by the processors that implement the PowerPC architecture. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. See Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*, for a list of simplified mnemonic examples.

Table 3-21. Branch Instructions

Name	Mnemonic	Operand Syntax
Branch	**b** (**ba** **bl** **bla**)	target_addr
Branch Conditional	**bc** (**bca** **bcl** **bcla**)	BO,BI,target_addr
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI

3.2.4.4.3 Condition Register Logical Instructions

Condition register logical instructions, shown in Table 3-22, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions, although they are executed by the system register unit (SRU). Most instructions executed by the SRU are completion-serialized to maintain system state; that is, the instruction is held for execution in the SRU until all prior instructions issued have completed.

Table 3-22. Condition Register Logical Instructions

Name	Mnemonic	Operand Syntax
Condition Register AND	**crand**	**crb**D,**crb**A,**crb**B
Condition Register AND with Complement	**crandc**	**crb**D,**crb**A,**crb**B
Condition Register Equivalent	**creqv**	**crb**D,**crb**A,**crb**B
Condition Register NAND	**crnand**	**crb**D,**crb**A,**crb**B
Condition Register NOR	**crnor**	**crb**D,**crb**A,**crb**B
Condition Register OR	**cror**	**crb**D,**crb**A,**crb**B
Condition Register OR with Complement	**crorc**	**crb**D,**crb**A,**crb**B
Condition Register XOR	**crxor**	**crb**D,**crb**A,**crb**B
Move Condition Register Field	**mcrf**	**crf**D,**crf**S

Note that if the LR update option is enabled for any of these instructions, these forms of the instructions are invalid in the e300 core.

3.2.4.5 Trap Instructions

The trap instructions shown in Table 3-23 are provided to test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap handler is invoked. If the tested conditions are not met, instruction execution continues normally.

Table 3-23. Trap Instructions

Name	Mnemonic	Operand Syntax
Trap Word	**tw**	TO,**r**A,**r**B
Trap Word Immediate	**twi**	TO,**r**A,SIMM

See Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*, for a complete set of simplified mnemonics.

3.2.4.6 Processor Control Instructions

UISA-level processor control instructions are used to read from and write to the condition register (CR).

3.2.4.6.1 Move To/From Condition Register Instructions

Table 3-24 lists the instructions provided by the core for reading from or writing to the CR.

Table 3-24. Move To/From Condition Register Instructions

Name	Mnemonic	Operand Syntax
Move from Condition Register	**mfcr**	**r**D
Move to Condition Register Fields	**mtcrf**	CRM,**r**S
Move to Condition Register from XER	**mcrxr**	**crf**D

3.2.4.7 Memory Synchronization Instructions—UISA

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events and the order in which memory operations are seen by other processors or memory access mechanisms. See Chapter 4, “Instruction and Data Cache Operation,” for additional information about these instructions and about related aspects of memory synchronization.

The **sync** instruction delays execution of subsequent instructions until previous instructions have completed to the point that they can no longer cause an interrupt and until all previous memory accesses are performed globally; the **sync** operation is not broadcast onto the e300 core Coherent System Bus (CSB) interface. Additionally, all load and store cache/bus activities initiated by prior instructions are completed. Touch load operations (icbt, **dcbt** and **dcbtst**) are required to complete at least through address translation but are not required to complete on the bus.

The functions performed by the **sync** instruction normally take a significant amount of time to complete; as a result, frequent use of this instruction may adversely affect performance. In addition, the number of cycles required to complete a **sync** instruction depends on system parameters and on the processor's state when the instruction is issued.

The proper paired use of the **lwarx** and **stwcx.** instructions allows programmers to emulate common semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Examples of these operations can be found in Appendix E, “Synchronization Programming Examples,” in the *Programming Environments Manual*. Typically, the **lwarx** instruction should be paired with an **stwcx.** instruction with the same effective address used for both instructions of the pair. Note that the reservation granularity is 32 bytes.

The concept behind the use of the **lwarx** and **stwcx.** instructions is that a processor may load a semaphore from memory, compute a result based on the value of the semaphore, and conditionally store it back to the same location (only if that location has not been modified since it was first read), and determine if the store was successful. The conditional store is performed, based on the existence of a reservation established by the preceding **lwarx** instruction. If the reservation exists when the store is executed, the store is performed which sets a bit in the CR. If the reservation does not exist when the store is executed, the target memory location is not modified and a bit is cleared in the CR.

If the store was successful, the sequence of instructions from the read of the semaphore to the store that updated the semaphore appear to have been executed atomically (that is, no other processor or mechanism modified the semaphore location between the read and the update), thus providing the equivalent of a real atomic operation. However, in reality, other cores may have read from the location during this operation. In the e300 core, the reservations are made on behalf of aligned 32-byte sections of the memory address space.

The **lwarx** and **stwcx.** instructions require the EA to be aligned. Interrupt-handling software should not attempt to emulate a misaligned **lwarx** or **stwcx.** instruction, because there is no correct way to define the address associated with the reservation.

In general, the **lwarx** and **stwcx.** instructions should be used only in system programs, which can be invoked by application programs as needed.

At most, one reservation exists simultaneously on any processor. The address associated with the reservation can be changed by a subsequent **lwarx** instruction. The conditional store is performed, based on the existence of a reservation established by the preceding **lwarx** regardless of whether the address generated by the **lwarx** matches that generated by the **stwcx.** instruction. A reservation held by the processor is cleared by one of the following:

- Executing an **stwcx.** instruction to any address
- Attempt by some other device to modify a location in the reservation granularity (32 bytes)

The **lwarx** and **stwcx.** instructions to write-through memory do not cause a DSI interrupt.

Table 3-25 lists the UISA memory synchronization instructions for the e300 core.

Table 3-25. Memory Synchronization Instructions—UISA

Name	Mnemonic	Operand Syntax
Load Word and Reserve Indexed	**lwarx**	**r**D,**r**A,**r**B
Store Word Conditional Indexed	**stwcx.**	**r**S,**r**A,**r**B
Synchronize	**sync**	—

3.2.5 PowerPC VEA Instructions

The VEA describes the semantics of the memory model that can be assumed by software processes, and includes descriptions of the cache model, cache-control instructions, address aliasing, and other related issues.

3.2.5.1 Processor Control Instructions

The VEA defines the Move from Time Base (**mftb**) instruction for reading the contents of the time base register. The **mftb** is a user-level instruction, as shown in Table 3-26.

Table 3-26. Move From Time Base Instruction

Name	Mnemonic	Operand Syntax
Move from Time Base	**mftb**	**r**D, TBR

Simplified mnemonics are provided for the **mftb** instruction so it can be coded with the TBR name as part of the mnemonic rather than requiring it to be coded as an operand. The **mftb** instruction serves as both a basic and simplified mnemonic. Assemblers recognize an **mftb** mnemonic with two operands as the basic form, and an **mftb** mnemonic with one operand as the simplified form. Simplified mnemonics are also provided for Move from Time Base Upper (**mftbu**), a variant of the **mftb** instruction rather than of **mfspr**. The core ignores the extended opcode differences between **mftb** and **mfspr** by ignoring bit 25 of both instructions and treating them identically. Refer to Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*.

3.2.5.2 Memory Synchronization Instructions—VEA

Memory synchronization instructions control the order in which memory operations are performed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. See Chapter 4, "Instruction and Data Cache Operation," for additional information about these instructions and about related aspects of memory synchronization.

Implementation Notes—The following describes how the core handles memory synchronization in the VEA.

- The Instruction Synchronize (**isync**) instruction causes the core to discard all prefetched instructions, wait for any preceding instructions to complete, and then branch to the next sequential instruction (having the effect of clearing the pipeline behind the **isync** instruction).
- The Enforce In-Order Execution of I/O (**eieio**) instruction is used to ensure memory reordering of noncacheable memory access. Because the core does not reorder noncacheable memory accesses, the **eieio** instruction is treated as a no-op.

Table 3-27 lists the VEA memory synchronization instructions for the core.

Table 3-27. Memory Synchronization Instructions—VEA

Name	Mnemonic	Operand Syntax
Enforce In-Order Execution of I/O	**eieio**	—
Instruction Synchronize	**isync**	—

3.2.5.3 Memory Control Instructions—VEA

Memory control instructions include the following types:

- Cache management instructions
- Segment register manipulation instructions
- Translation lookaside buffer management instructions

This section describes the user-level cache management instructions defined by the VEA. See Section 3.2.6.3, "Memory Control Instructions—OEA," for information about supervisor-level cache, segment register manipulation, and translation lookaside buffer management instructions.

The instructions listed in Table 3-28 provide user-level programs the ability to manage on-chip caches when they exist.

Table 3-28. User-Level Cache Instructions

Name	Mnemonic	Operand Syntax
Data Cache Block Flush	**dcbf**	**r**A,**r**B
Data Cache Block Set to Zero	**dcbz**	**r**A,**r**B
Data Cache Block Store	**dcbst**	**r**A,**r**B
Data Cache Block Touch	**dcbt**	**r**A,**r**B
Data Cache Block Touch for Store	**dcbtst**	**r**A,**r**B
Instruction Cache Block Invalidate	**icbi**	**r**A,**r**B

As with other memory-related instructions, the effect of the cache management instructions on memory are weakly ordered. If the programmer needs to ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, a **sync** instruction must be placed in the program following those instructions.

Note that when data address translation is disabled (MSR[DR] = 0), the Data Cache Block Set to Zero (**dcbz**) instruction allocates a cache block in the cache and may not verify that the physical address is valid. If a cache block is created for an invalid physical address, a machine check condition may result when an attempt is made to write that cache block back to memory. The cache block could be written back as a result of the execution of an instruction that causes a cache miss and the invalid addressed cache block is the target for replacement or a Data Cache Block Store (**dcbst**) instruction.

Table 3-28 lists the cache instructions that are accessible to user-level programs.

3.2.6 PowerPC OEA Instructions

The OEA includes the structure of the memory management model, supervisor-level registers, and interrupt model.

3.2.6.1 System Linkage Instructions

This section describes the system linkage instructions (see Table 3-29). The **sc** instruction is a user-level instruction that permits a user program to call on the system to perform a service and causes the processor to take an interrupt. The Return from Interrupt (**rfi**) instruction is a supervisor-level instruction that is useful for returning from an interrupt handler.

The Return from Critical Interrupt (**rfci**) instruction is a supervisor-level instruction that is implemented in the core. The **rfci** instruction is useful for returning from a critical interrupt handler. This instruction is described in Section 3.2.8, “Implementation-Specific Instructions.”

Table 3-29. System Linkage Instructions

Name	Mnemonic	Operand Syntax
Return from Interrupt	**rfi**	—
System Call	**sc**	—

3.2.6.2 Processor Control Instructions—OEA

Processor control instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs), and to read from the time base register (TBU or TBL).

3.2.6.2.1 Move To/From Machine State Register Instructions

Table 3-30 lists the instructions provided by the core for reading from or writing to the MSR.

Table 3-30. Move To/From Machine State Register Instructions

Name	Mnemonic	Operand Syntax
Move from Machine State Register	**mfmsr**	**r**D
Move to Machine State Register	**mtmsr**	**r**S

3.2.6.2.2 Move To/From Special-Purpose Register Instructions

Simplified mnemonics are provided for the **mtspr** and **mfspr** instructions so they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. See Appendix F, "Simplified Mnemonics," in the *Programming Environments Manual*, for simplified mnemonic examples. The **mtspr** and **mfspr** instructions are shown in Table 3-31.

Table 3-31. Move To/From Special-Purpose Register Instructions

Name	Mnemonic	Operand Syntax
Move from Special-Purpose Register	**mfspr**	**r**D,SPR
Move to Special-Purpose Register	**mtspr**	SPR,**r**S

For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction encoding, with the high-order 5 bits appearing in bits 16–20 of the instruction encoding and the low-order 5 bits in bits 11–15.

If the SPR field contains any value other than one of the values shown in Table 3-32, either the program interrupt handler is invoked or the results are boundedly undefined.

Table 3-32. Implementation-Specific SPR Encodings (mfspr)

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
1	00000	00001	XER	User
8	00000	01000	LR	User
9	00000	01001	CTR	User
18	00000	10010	DSISR	Supervisor
19	00000	10011	DAR	Supervisor
22	00000	10110	DEC	Supervisor
25	00000	11001	SDR1	Supervisor
26	00000	11010	SRR0	Supervisor
27	00000	11011	SRR1	Supervisor
58	00001	11010	CSRR0[2]	Supervisor
59	00001	11011	CSRR1 [2]	Supervisor

Table 3-32. Implementation-Specific SPR Encodings (mfspr) (continued)

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
272	01000	10000	SPRG0	Supervisor
273	01000	10001	SPRG1	Supervisor
274	01000	10010	SPRG2	Supervisor
275	01000	10011	SPRG3	Supervisor
276	01000	10100	SPRG4 [2]	Supervisor
277	01000	10101	SPRG5 [2]	Supervisor
278	01000	10110	SPRG6 [2]	Supervisor
279	01000	10111	SPRG7 [2]	Supervisor
284	01000	11100	TBL	Supervisor
285	01000	11101	TBU	Supervisor
286	01000	11110	SVR [2]	Supervisor
287	01000	11111	PVR	Supervisor
309	01001	10101	IBCR [2]	Supervisor
310	01001	10110	DBCR [2]	Supervisor
311	01001	10111	MBAR [2]	Supervisor
317	01001	11101	DABR2 [2]	Supervisor
528	10000	10000	IBAT0U	Supervisor
529	10000	10001	IBAT0L	Supervisor
530	10000	10010	IBAT1U	Supervisor
531	10000	10011	IBAT1L	Supervisor
532	10000	10100	IBAT2U	Supervisor
533	10000	10101	IBAT2L	Supervisor
534	10000	10110	IBAT3U	Supervisor
535	10000	10111	IBAT3L	Supervisor
536	10000	11000	DBAT0U	Supervisor
537	10000	11001	DBAT0L	Supervisor
538	10000	11010	DBAT1U	Supervisor
539	10000	11011	DBAT1L	Supervisor
540	10000	11100	DBAT2U	Supervisor
541	10000	11101	DBAT2L	Supervisor
542	10000	11110	DBAT3U	Supervisor
543	10000	11111	DBAT3L	Supervisor
560	10001	10000	IBAT4U [2]	Supervisor

Table 3-32. Implementation-Specific SPR Encodings (mfspr) (continued)

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
561	10001	10001	IBAT4L [2]	Supervisor
562	10001	10010	IBAT5U [2]	Supervisor
563	10001	10011	IBAT5L [2]	Supervisor
564	10001	10100	IBAT6U [2]	Supervisor
565	10001	10101	IBAT6L [2]	Supervisor
566	10001	10110	IBAT7U [2]	Supervisor
567	10001	10111	IBAT7L [2]	Supervisor
568	10001	11000	DBAT4U [2]	Supervisor
569	10001	11001	DBAT4L [2]	Supervisor
570	10001	11010	DBAT5U [2]	Supervisor
571	10001	11011	DBAT5L [2]	Supervisor
572	10001	11100	DBAT6U [2]	Supervisor
573	10001	11101	DBAT6L	Supervisor
574	10001	11110	DBAT7U [2]	Supervisor
575	10001	11111	DBAT7L [2]	Supervisor
976	11110	10000	DMISS	Supervisor
977	11110	10001	DCMP	Supervisor
978	11110	10010	HASH1	Supervisor
979	11110	10011	HASH2	Supervisor
980	11110	10100	IMISS	Supervisor
981	11110	10101	ICMP	Supervisor
982	11110	10110	RPA	Supervisor
1008	11111	10000	HID0	Supervisor
1009	11111	10001	HID1	Supervisor
1010	11111	10010	IABR	Supervisor
1011	11111	10011	HID2	Supervisor
1013	11111	10101	DABR [2]	Supervisor
1018	11111	11010	IABR2 [2]	Supervisor

[1] Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.

For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16–20 of the instruction and the low-order 5 bits in bits 11–15.

[2] These registers are implementation-specific for e300 core only.

Implementation note—The core ignores the extended opcode differences between **mftb** and **mfspr** by ignoring TB[25] and treating both instructions identically.

3.2.6.2.3 Move To/From Performance Monitor Register Instructions

The APU defines instructions for reading and writing the PMRs as shown in Table 3-33.

Table 3-33. Performance Monitor APU Instructions

Name	Mnemonic	Syntax
Move from Performance Monitor Register	**mfpmr**	**r**D,PMRN
Move to Performance Monitor Register	**mtpmr**	PMRN,**r**S

3.2.6.3 Memory Control Instructions—OEA

This section describes memory control instructions, which include the following types:

- Cache management instructions
- Segment register manipulation instructions
- Translation lookaside buffer management instructions

3.2.6.3.1 Supervisor-Level Cache Management Instruction

The supervisor-level cache management instruction in the PowerPC architecture, **dcbi**, is used to invalidate individual cache blocks.The **icbt** instruction, another supervisor-level instruction, performs a bus read operation from the bus and allocates into the instruction cache. This instruction is new to the e300 core, and supplements the instruction cache locking mechanisms and the new lock-protect feature. The user-level **dcbf** instruction, described in Section 3.2.5.3, “Memory Control Instructions—VEA,” and Section 4.5.2, “Cache Control Instructions,” should be used when the program needs to invalidate cache blocks. Note that the **dcbf** instruction causes modified blocks to be flushed to system memory if they are the target of a **dcbf** instruction, whereas, by definition in the PowerPC architecture, the **dcbi** instruction only invalidates modified blocks.

3.2.6.3.2 Segment Register Manipulation Instructions

The instructions listed in Table 3-34 provide access to the segment registers for the e300 core. These instructions operate completely independently from the MSR[IR] and MSR[DR] bit settings. Refer to “Synchronization Requirements for Special Registers and TLBs” in Chapter 2, “Register Set,” in the *Programming Environments Manual*, for serialization requirements and other recommended precautions to observe when manipulating the segment registers.

Table 3-34. Segment Register Manipulation Instructions

Name	Mnemonic	Operand Syntax
Move from Segment Register	**mfsr**	**r**D,SR
Move from Segment Register Indirect	**mfsrin**	**r**D,**r**B

Table 3-34. Segment Register Manipulation Instructions (continued)

Name	Mnemonic	Operand Syntax
Move to Segment Register	**mtsr**	SR,**r**S
Move to Segment Register Indirect	**mtsrin**	**r**S,**r**B

3.2.6.3.3 Translation Lookaside Buffer Management Instructions

The address translation mechanism is defined in terms of segment descriptors and page table entries (PTEs) used by the processors to locate the effective-to-physical address mapping for a particular access. The PTEs reside in page tables in memory. As defined for 32-bit implementations by the PowerPC architecture, segment descriptors reside in 16 on-chip segment registers.

The e300 core provides the ability to invalidate a TLB entry. The TLB Invalidate Entry (**tlbie**) instruction invalidates the TLB entry indexed by the EA, and operates on both the instruction and data TLBs simultaneously invalidating four TLB entries (both sets in each TLB). The index corresponds to bits 15–19 of the EA. To invalidate all entries within both TLBs, 32 **tlbie** instructions should be issued, incrementing this field by one each time.

The core provides two implementation-specific instructions (**tlbld** and **tlbli**) that are used by software table search operations following TLB misses to load TLB entries on-chip.

For more information on **tlbld** and **tlbli** refer to Section 3.2.8, “Implementation-Specific Instructions.”

Note that the **tlbia** instruction is not implemented on the core.

Refer to Chapter 6, “Memory Management,” for more information about the TLB operations for the core. Table 3-35 lists the TLB instructions.

Table 3-35. Translation Lookaside Buffer Management Instructions

Name	Mnemonic	Operand Syntax
Load Data TLB Entry	**tlbld**	**r**B
Load Instruction TLB Entry	**tlbli**	**r**B
TLB Invalidate Entry	**tlbie**	**r**B
TLB Synchronize	**tlbsync**	—

Because the presence and exact semantics of the translation lookaside buffer management instructions is implementation-dependent, system software should incorporate uses of the instructions into subroutines to maximize compatibility with programs written for other processors.

For more information on the PowerPC instruction set, refer to Chapter 4, “Addressing Modes and Instruction Set Summary,” and Chapter 8, “Instruction Set,” in the *Programming Environments Manual*.

3.2.7 Recommended Simplified Mnemonics

To simplify assembly language programs, a set of simplified mnemonics is provided for some of the most frequently used operations (such as no-op, load immediate, load address, move register, and complement register). PowerPC compliant assemblers provide the simplified mnemonics listed in “Recommended

Simplified Mnemonics” in Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*, and listed with some of the instruction descriptions in this chapter. Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in this document.

For a complete list of simplified mnemonics, see Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*.

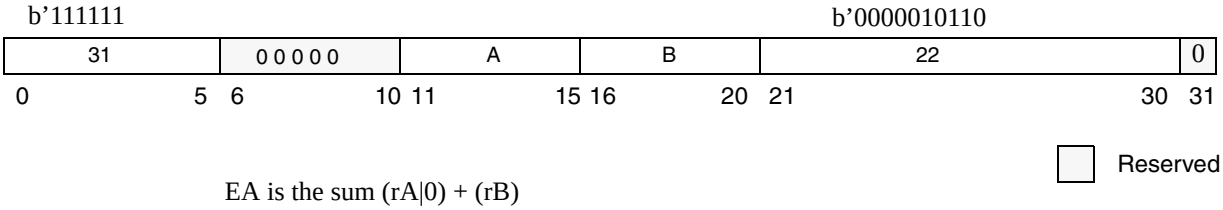
3.2.8 Implementation-Specific Instructions

This section provides a detailed look at the e300 core implementation-specific instructions—**icbt, tlbld**, **tlbli**, **and rfci**.

icbt
Instruction Cache Block Touch

icbt **rA, rB**

icbt
Integer Unit



EA is the sum (rA|0) + (rB)

The instruction is a hint that the performance will be improved if the block containing the instruction addressed by the EA is fetched into the instruction cache, because the program will execute from the addressed location. Executing an **icbt** instruction does not cause any interrupt to be invoked.

If the EA specifies a location outside of the main memory, the instruction is treated as a no-op. Also, the **icbt** instruction is treated as a no-op if touch load operation is disabled by the HID0[NOPTI] configuration bit. The **icbt** instruction is effective regardless of WIMG settings, instruction or data cache enable status, or the instruction cache lock status.

This is a supervisor-level, context synchronizing instruction. It is e300-specific and not part of the PowerPC instruction set.

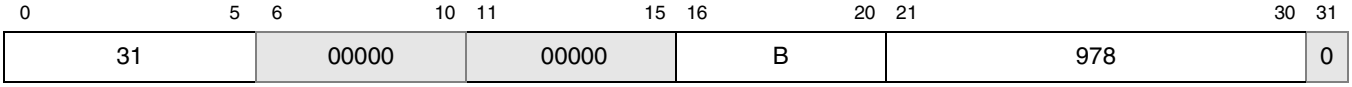
Other registers altered:

- None

tlbld tlbld

Load Data TLB Entry Integer Unit

tlbld **r**B




```
EA ← (rB)
TLB entry created from DCMP and RPA
DTLB entry selected by EA[15-19] and SRR1[WAY] ← created TLB entry
```

The EA is the contents of **r**B. The **tlbld** instruction loads the contents of the data PTE compare (DCMP) and required physical address (RPA) registers into the first word of the selected data TLB entry. The specific DTLB entry to be loaded is selected by the EA and SRR1[WAY] bit.

The **tlbld** instruction should only be executed when address translation is disabled (MSR[IR] = 0 and MSR[DR] = 0).

Note that it is possible to execute the **tlbld** instruction when address translation is enabled; however, extreme caution should be used in doing so. If data address translation is set (MSR[DR] = 1) **tlbld** must be preceded by a **sync** instruction and succeeded by a context synchronizing instruction.

Also, note that care should be taken to avoid modification of the instruction TLB entries that translate current instruction prefetch addresses.

This is a supervisor-level instruction; it is also an e300 core-specific instruction, and not part of the PowerPC instruction set.

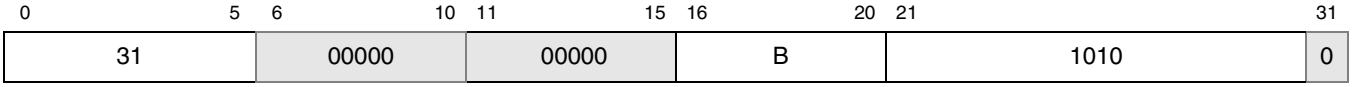
Other registers altered:

- None

tlbli tlbli

Load Instruction TLB Entry Integer Unit

tlbld **r**B




```
EA ← (rB)
TLB entry created from ICMP and RPA
ITLB entry selected by EA[15-19] and SRR1[WAY] ← created TLB entry
```

The EA is the contents of **r**B. The **tlbli** instruction loads the contents of the instruction PTE compare (ICMP) and required physical address (RPA) registers into the first word of the selected instruction TLB entry. The specific ITLB entry to be loaded is selected by the EA and SRR1[WAY] bit.

The **tlbli** instruction should only be executed when address translation is disabled (MSR[IR] = 0 and MSR[DR] = 0).

Note that it is possible to execute the **tlbld** instruction when address translation is enabled; however, extreme caution should be used in doing so. If instruction address translation is set (MSR[IR] = 1), **tlbli** must be followed by a context synchronizing instruction such as **isync** or **rfi**.

Also, note that care should be taken to avoid modification of the instruction TLB entries that translate current instruction prefetch addresses.

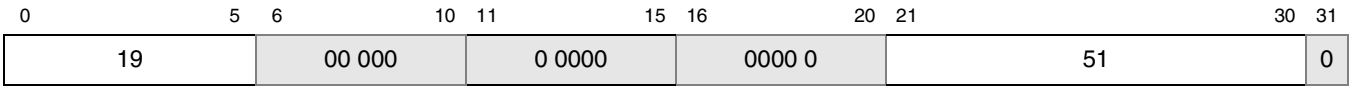
This is a supervisor-level instruction; it is also an e300 core-specific instruction, and not part of the PowerPC instruction set.

Other registers altered:

- None

rfci rfci

Return from Critical Interrupt




```
MSR[16–27, 30–31] ← CSRR1[16–27, 30–31]
NIA ←iea CSRR0[0–29] || 0b00
```

Bits CSRR1[16-27, 30–31] are placed into the corresponding bits of the MSR. If the new MSR value does not enable any pending interrupts, then the next instruction is fetched, under control of the new MSR value, from the address CSRR0[0–29] || 0b00. If the new MSR value enables one or more pending interrupts, the interrupt associated with the highest priority pending interrupt is generated; in this case the value placed into CSRR0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred. Note that an implementation may define additional MSR bits, and in this case, may also cause them to be saved to CSRR1 from MSR on an interrupt and restored to MSR from CSRR1 on an **rfci**.

This is a supervisor-level, context synchronizing instruction. This instruction is defined only for 32-bit implementations.

Other registers altered:

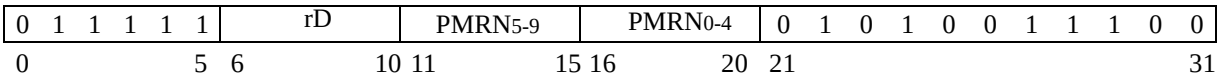
- MSR

The following instruction has been added to support performance monitor operations:

mfpmr
Move From Performance Monitor Register

mfpmr **rD, PMRN**

mfpmr
Integer Unit



GPR (rD) <-- PMREG(PMRN)

PMRN denotes a performance monitor register as listed in Table 11-1 and Table 11-2.

The contents of the designated performance monitor register are placed into GPR[rD].

When MSR[PR] = 1, specifying a performance monitor register that is not implemented and is not privileged (PMRN[5] = 0) results in an illegal instruction exception-type program interrupt. When MSR[PR]=1, specifying a performance monitor register that is privileged (PMRN[5] = 1) results in a privileged instruction execution-type program interrupt. When MSR[PR] = 0, specifying an unimplemented performance monitor register is boundedly undefined.

Other registers altered:

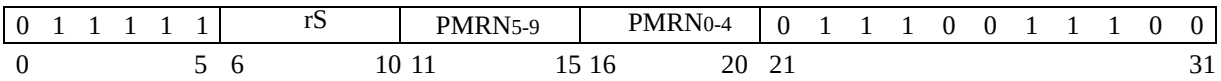
none

mtpmr

Move To Performance Monitor Register

mtpmr **PMRN, rS**

mtpmr
Integer Unit



PMREG(PMRN) <-- GPR (rS)

PMRN denotes a performance monitor register, as listed in Table 11-1 and Table 11-2.

The contents of GPR[rS] are placed into the designated performance monitor register.

When MSR[PR] = 1, specifying a performance monitor register that is not implemented and is not privileged (PMRN[5] = 0) results in an illegal instruction exception-type program interrupt. When MSR[PR]=1, specifying a performance monitor register that is privileged (PMRN[5] = 1) results in a privileged instruction execution-type program interrupt. When MSR[PR] = 0, specifying an unimplemented performance monitor register is boundedly undefined.

Other registers altered:

none

Chapter 4
Instruction and Data Cache Operation

This chapter describes the organization of the cache, cache coherency protocols, cache control instructions, and various cache operations. It describes the interaction between the caches, the load/store unit, the instruction unit, and the memory subsystem. It also describes the cache way-locking features provided in the core.

Note that in this chapter the term ‘multiprocessor’ is used in the context of maintaining cache coherency. These multiprocessor devices could be actual processors and other devices that can access system memory, maintain their own caches, and function as bus masters requiring cache coherency.

4.1 Introduction

The core provides two independent L1 instruction and data caches to allow the registers and execution units rapid access to instructions and data.

4.1.1 Instruction and Data Cache Features

The cache implementation has the following characteristics:

- Harvard architecture—separate instruction and data caches
- 32-Kbyte instruction and data caches on e300c1; 16-Kbyte caches on e300c2 and e300c3
- Eight-way, set-associativity on e300c1, and four-way set-associative on e300c2 and e300c3
- Physically addressed cache directories. The physical (real) address tag is stored in the cache directory.
- 32-byte cache blocks. A cache block is the block of memory that a coherency state describes, also referred to as a cache line.
- A four-state modified/exclusive/shared/invalid (MESI), or a three-state modified/exclusive/invalid (MEI) coherency protocol for the data cache
- Two status bits for each data cache block to indicate the coherency state as follows:
  - Modified (M)
  - Exclusive (E)
  - Shared (S)
  - Invalid (I)
- A single status bit for each instruction cache block that allows encoding for the following two possible states:
  - Invalid
  - Valid

- Cache locking:
  - Entire cache locking for each cache by setting the appropriate bits in the HID0
  - Way-locking support for instruction and data caches using controls in HID2
- Cache invalidation by setting the appropriate bits in HID0
- A pseudo least-recently-used (PLRU) replacement algorithm within each set
- An instruction cancel mechanism that improves use of instruction cache by supporting hits-under-cancels and misses-under-cancels
- Data cache queue sharing that makes cast-outs and snoop pushes more efficient
- New **icbt** instruction supports initializing instruction cache
- An instruction fetch burst feature that allows all instruction fetches from caching-inhibited space to be performed on the bus as burst transactions
- Parity support for both instruction and data caches

4.1.2 Overview

The core supports a fully coherent, 4-Gbyte physical memory address space. Bus snooping is used to ensure the coherency of global memory with respect to the data cache.

On a cache miss, cache blocks are loaded in four beats of 64 bits each when the core is configured for a 64-bit data bus; when the core is configured for a 32-bit bus, cache block loads are performed as eight beats of 32 bits each. Regardless of the bus size, the burst load is performed as a critical-double-word-first operation.

For data cache loads, the data cache is blocked to internal accesses until the load completes. The critical-double-word is written to the cache and forwarded to the requesting unit, minimizing stalls due to load delays. Note that the cache line being filled cannot be accessed internally until the fill completes.

The instruction cache allows sequential fetching during a cache block load; the instruction cache is blocked only until the cache line load completes.

Both caches are tightly coupled to the core bus interface unit (BIU) to allow efficient access to the system memory controller and other bus masters. The core load/store unit (LSU) is also directly coupled to the data cache to allow the efficient movement of data to and from the general-purpose and floating-point registers. The core bus interface unit receives requests for bus operations from the instruction and data caches, and executes the operations according to the coherent system bus (CSB) protocol. The BIU provides transaction queuing, prioritization, and bus control logic. The BIU also captures snoop addresses for data cache, address queue, and memory reservation (**lwarx** and **stwcx**. instruction) operations. The instruction cache is not snooped; therefore, instruction cache coherency must be maintained by software.

The LSU provides the data transfer interface between the data cache and the GPRs and FPRs. It provides all logic required to calculate effective addresses, handle data alignment to and from the data cache, and provides sequencing for load and store string and multiple operations. As shown in Figure 1-1, the caches provide a 64-bit interface to the instruction fetcher and LSU. Write operations to the data cache can be performed on a byte, half-word, word, or double-word basis.

4.2 Data Cache Organization

The e300c1 data cache is configured as 128 sets of eight blocks per set. The organization of the e300c1 data cache is shown in Figure 4-1.

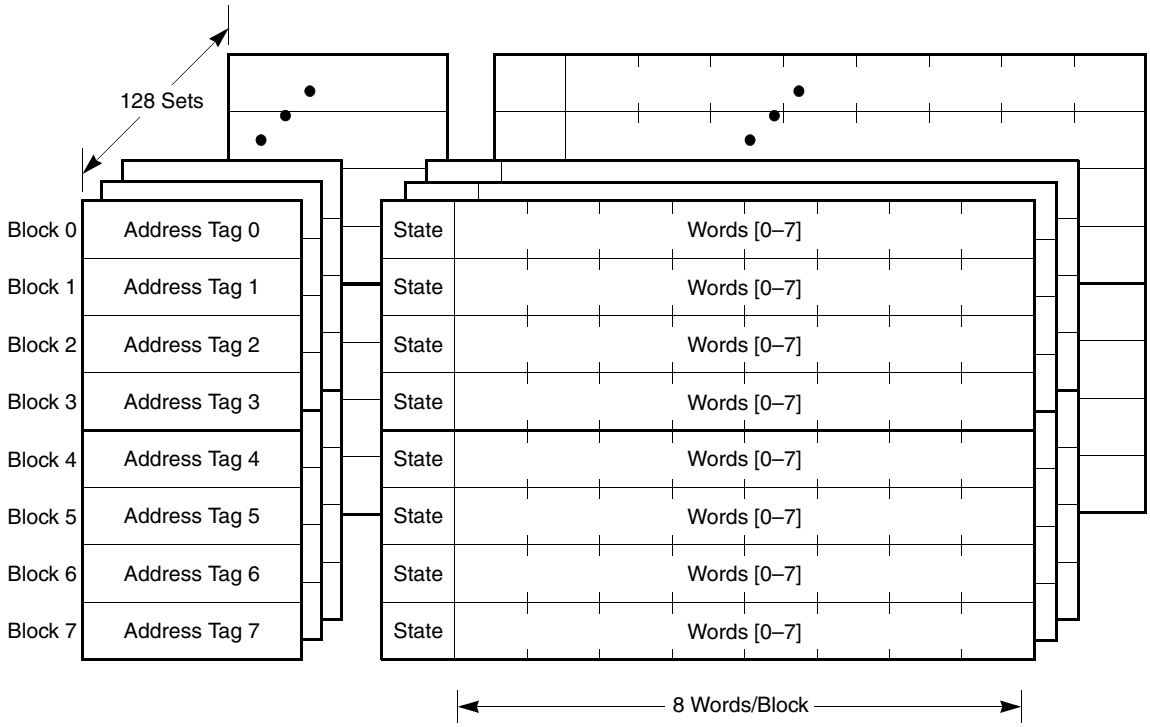


Figure 4-1. e300c1 Data Cache Organization

The e300c2 and e300c3 data cache is configured as 128 sets of four blocks per set. The organization of the e300c2 and e300c3 data cache is shown in Figure 4-2.

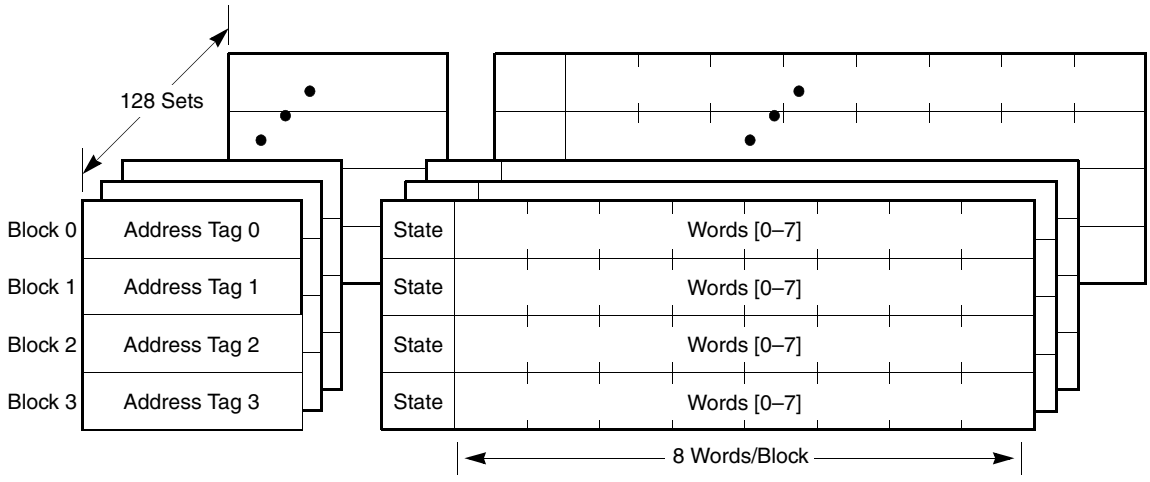


Figure 4-2. e300c2 and e300c3 Data Cache Organization

Each block consists of 32 bytes of data, 32 parity bits, state bits, and an address tag. Each cache block contains eight contiguous words from memory that are loaded from an eight-word boundary (that is, bits

A27–A31 are not part of the cache block address); thus, a cache block never crosses a page boundary. Misaligned accesses across a page boundary can incur a performance penalty.

Bits A20–A26 provide an index to select a set. Bits A27–A31 select a byte within a block. The tags consists of bits PA0–PA19. Address translation occurs in parallel, such that higher-order bits (the tag bits in the cache) are physical.

The state bits implement either a standard, four-state MESI coherency protocol or a three-state MEI protocol. Cache coherency is enforced by on-chip bus snooping logic. Since the data cache tags are single-ported, a load or store, simultaneous with a snoop access, represents a resource contention. The snoop access is given first access to the tags. Load or store operations can be performed to the cache on the clock cycle immediately following a snoop access if the snoop misses. Snoop hits may block the data cache for two or more cycles, depending on whether a copy back to main memory is required. The replacement algorithm is a PLRU algorithm; that is, the pseudo least-recently used block is filled with new data on a cache miss.

4.3 Instruction Cache Organization

The e300c1 instruction cache also consists of 128 sets of eight blocks per set. The e300c1 instruction cache organization is shown in Figure 4-3.

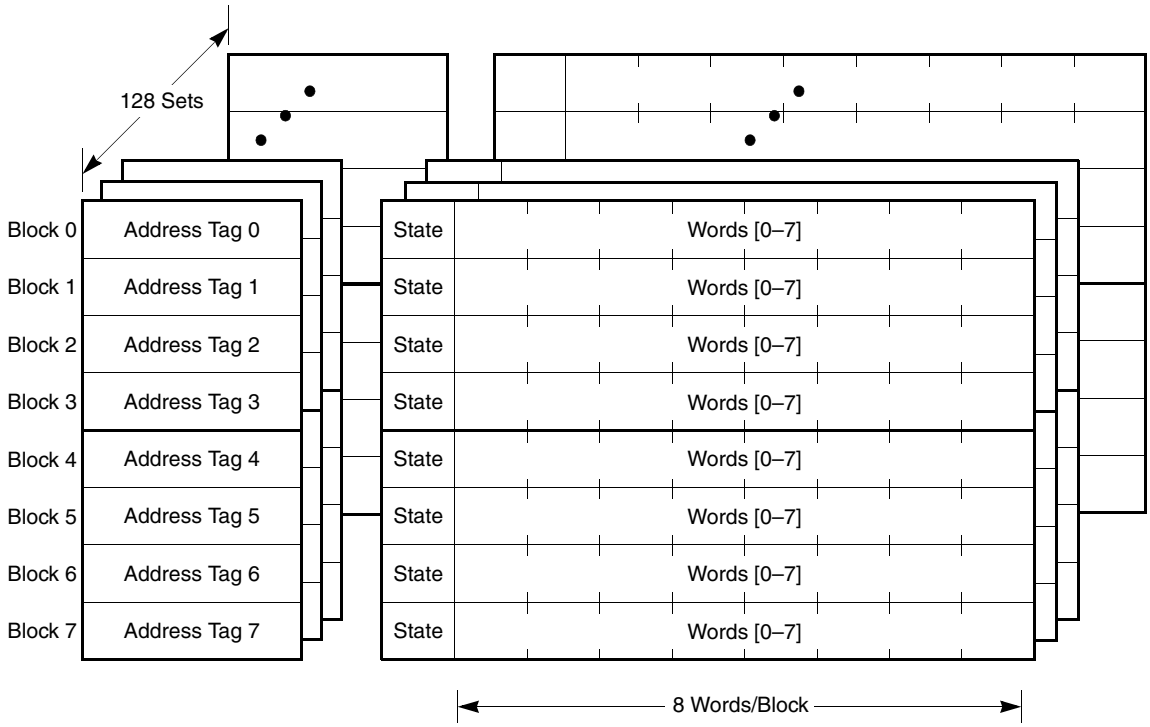


Figure 4-3. e300c1 Instruction Cache Organization

The e300c2 and e300c3 instruction cache is configured as 128 sets of four blocks per set. The organization of the e300c2 and e300c3 instruction cache is shown in Figure 4-4.

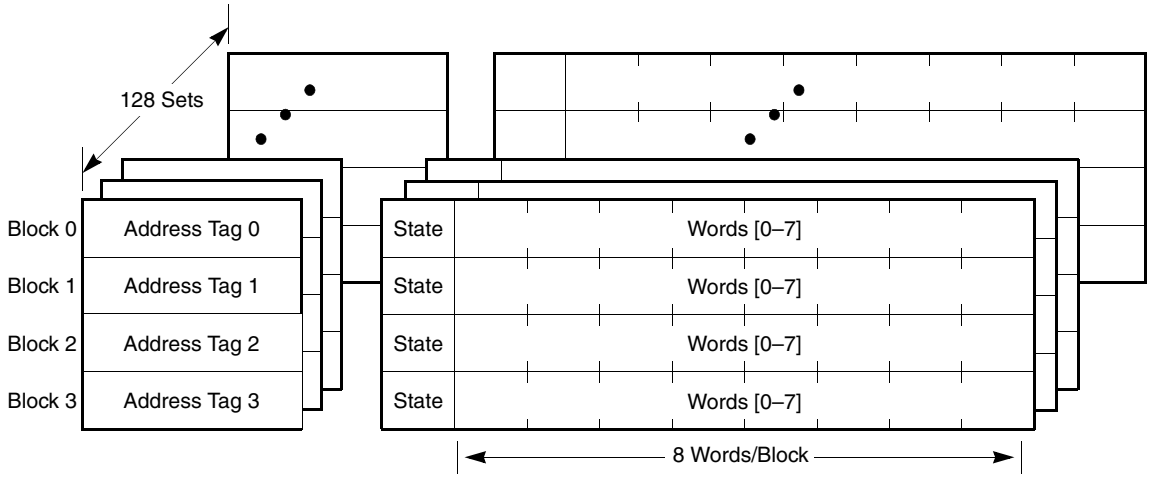


Figure 4-4. e300c2 and e300c3 Instruction Cache Organization

Each block consists of 32 bytes of data, 32 parity bits, an address tag, and a valid bit. Each cache block contains eight contiguous words from memory that are loaded from an eight-word boundary (that is, bits A27–A31 are not part of the cache block address); thus, a cache block never crosses a page boundary. Misaligned accesses across a page boundary can incur a performance penalty.

Address bits A20–A26 provide an index to select a set. Bits A27–A31 select a byte within a block. The tags consists of bits PA0–PA19. Address translation occurs in parallel, such that higher-order bits (the tag bits in the cache) are physical. The replacement algorithm is a PLRU algorithm; that is, the pseudo least-recently used block is filled with new instructions on a cache miss.

The instruction cache is only written as a result of a block fill operation on a cache miss. A hardware invalidation capability is provided to support cache maintenance. The instruction fetcher accesses the instruction cache frequently in order to sustain the high throughput provided by the six-entry instruction queue.

4.4 Memory and Cache Coherency

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. Coherency allows synchronization and cooperative use of shared resources. Otherwise, multiple copies of a memory location, some containing stale values, could exist in a system, resulting in errors when the stale values are used. This section describes the coherency mechanisms of the PowerPC architecture and the cache coherency protocol that the data cache supports. Note that, unless specifically noted, the discussion of coherency in this section applies to the data cache only. The instruction cache is not snooped. Instruction cache coherency must be maintained by software.

4.4.1 Memory/Cache Access Attributes (WIMG Bits)

Some memory characteristics can be set on either a block or page basis by using the WIMG bits in the BAT registers or page table entry (PTE), respectively. The WIMG attributes control the following functionality:

- Write-through (W bit)
- Caching-inhibited (I bit)
- Memory coherency (M bit)
- Guarded memory (G bit)

The WIMG attributes are programmed by the operating system for each page and block. The W and I attributes control how the processor performing an access uses its own cache. The M attribute ensures that coherency is maintained for all copies of the addressed memory location. The G attribute prevents speculative (referred to as ‘out-of-order’ in the architecture specification) loading and prefetching from the addressed memory location. These bits allow both uniprocessor and multiprocessor system designs to exploit numerous system-level performance optimizations.

For accesses performed with real addressing mode (MSR[IR] = 0 or MSR[DR] = 0 for instruction or data access, respectively), the WIMG bits are automatically generated as 0b0011 (the data is write-back, caching is enabled, memory coherency is enforced, and memory is guarded).

Careless use of these bits may create situations where coherency paradoxes are observed by the processor. In particular, this can happen when the state of these bits is changed without appropriate precautions. For example, when flushing the pages that correspond to the changed bits from the caches of all processors in the system is required, or when the address translations of aliased physical addresses (referred to as real addresses in the architecture specification), specify different values for any of the WIM bits). The core considers any of these cases to be a programming error that may compromise the coherency of memory. These paradoxes can occur within a single processor or across several devices, as described in Section 4.4.2.4, “Coherency in Single-Processor Systems.”

4.4.1.1 Write-Through Attribute (W)

When an access is designated as write-through (W = 1), if the data is in the cache, a store operation updates the cached copy of the data. In addition, the update is written to the external memory location (as described below).

While the PowerPC architecture permits multiple store instructions to be combined for write-through accesses except when the store instructions are separated by a **sync** or **eieio** instruction, the e300 core does not implement this combined-store capability. A store operation that uses the write-through attribute may cause any part of valid data in the cache to be written back to main memory. The **eieio** instruction is treated as a no-op by the e300 core.

The definition of the external memory location to be written to, in addition to the on-chip cache, depends on the implementation of the memory system and can be illustrated by the following examples:

- RAM—The store is sent to the RAM controller to be written into the target RAM.
- I/O device—The store is sent to the memory-mapped I/O control hardware to be written to the target register or memory location.

In systems with multilevel caching, the store must be written to at least a depth in the memory hierarchy that is seen by all processors and devices.

Accesses that correspond to W = 0 are considered write-back. For this case, although the store operation is performed to the cache, it is only made to external memory when a copy-back operation is required. Use of the write-back mode (W = 0) can improve overall performance for areas of the memory space that are seldom referenced by other masters in the system.

4.4.1.2 Caching-Inhibited Attribute (I)

If I = 1, the memory access is completed by referencing the location in main memory, bypassing the on-chip cache. During the access, the addressed location is not loaded into the cache, nor is the location allocated in the cache. It is considered a programming error if a copy of the target location of an access to caching-inhibited memory is resident in the cache. Software must ensure that the location has not been previously loaded into the cache, or, if it has, that it has been flushed from the cache.

The PowerPC architecture permits data accesses from more than one instruction to be combined for caching-inhibited operations, except when the accesses are separated by a **sync** instruction or by an **eieio** instruction when the page or block is also designated as guarded. This combined-access capability is not implemented on the e300 core. The **eieio** instruction is treated as a no-op by the e300 core.

The caching-inhibited (I) bit in the core also controls whether load and store operations are strongly or weakly ordered. If an I/O device requires load and store accesses to occur in program order, then the I bit for the page must be set. Refer to Section 4.4.3.2, “Sequential Consistency of Memory Accesses,” for more information.

4.4.1.3 Memory Coherency Attribute (M)

When an access requires coherency, the processor performing the access must inform the coherency mechanisms throughout the system that the access requires memory coherency. The M attribute determines the kind of access performed on the bus (global or local). This attribute is provided to allow improved performance in systems where hardware-enforced coherency is relatively slow and where software is able to enforce the required coherency. When M = 0, the processor does not enforce data coherency. When M = 1, the processor enforces data coherency, and the corresponding access is considered to be a global access.

When the M attribute is set, and the access is performed, the global signal is asserted to indicate that the access is global. Snooping devices affected by the access must then respond to this global access if their data is modified by signaling retry and by updating the memory location.

For instruction accesses, HID0[IFEM] control how accesses are performed (global or local) on the bus. If translation is enabled, setting IFEM causes the core to reflect the M bit state on the bus during instruction fetches.

4.4.1.4 Guarded Attribute (G)

When the guarded bit is set, the memory area (block or page) is designated as guarded. This setting can be used to protect certain memory areas from read accesses made by the processor that are not dictated

directly by the program. If there are areas of memory that are not fully populated (in other words, there are holes in the memory map within this area), this setting can protect the system from undesired accesses caused by speculative load operations or instruction prefetches that could lead to the generation of the machine check interrupt. Also, the guarded bit can be used to prevent speculative load operations or prefetches from occurring to certain peripheral devices where such speculative accesses could produce undesired results (for example, a destructive read from a FIFO buffer).

The processor will perform speculative out-of-order accesses to this area of memory, only as follows:

- Speculative load operations from guarded memory areas are performed only if the corresponding data is resident in the cache.
- The processor prefetches from guarded areas, but only when required, and only within the memory boundary dictated by the cache block. That is, if an instruction is certain to be required for execution by the program, it is fetched and the remaining instructions in the block may be prefetched, even if the area is guarded.

4.4.1.5 W, I, and M Bit Combinations

Table 4-1 summarizes the six combinations of the WIM bits. A setting of zero or one for the G bit is allowed for each of these WIM bit combinations.

Table 4-1. Combinations of W, I, and M Bits

WIM Setting	Meaning
000	Data may be cached. Loads or stores whose target hits in the cache use that entry in the cache. Memory coherency is not enforced by hardware.
001	Data may be cached. Loads or stores whose target hits in the cache use that entry in the cache. Memory coherency is enforced by hardware.
*n*10	Caching is inhibited. The access is performed to external memory, completely bypassing the cache. Memory coherency is not enforced by hardware.
*n*11	Caching is inhibited. The access is performed to external memory, completely bypassing the cache. Memory coherency must be enforced by external hardware (processor provides hardware indication that access is global).
100	Data may be cached. Load operations whose target hits in the cache use that entry in the cache. Stores are written to external memory. The target location of the store may be cached and is updated on a hit. Memory coherency is not enforced by hardware.
101	Data may be cached. Load operations whose target hits in the cache use that entry in the cache. Stores are written to external memory. The target location of the store may be cached and is updated on a hit. Memory coherency is enforced by hardware.

4.4.2 Coherency Support

The data cache of the e300 core supports both the three-state MEI and four-state MESI cache coherency protocols, as selected by the HID2[MESI] control parameter. For these modes, each 32-byte block in the data cache is always in one of the following states: modified (M), exclusive (E), shared (S), or invalid (I). The shared state is only available in MESI protocol mode. The instruction cache always operates in a reduced two-state mode using the states: valid (V) or invalid (I). Cache blocks in the instruction cache are never modified.

The four MESI states are defined in Table 4-2.

Table 4-2. MEI/MESI State Definitions

MESI State	Definition
Modified (M)	The addressed cache block is valid only in the cache. The cache block is modified with respect to system memory—that is, the modified data in the cache block has not been written back to memory.
Exclusive (E)	The addressed block is in this cache only. The data in this cache block is consistent with system memory.
Shared (S)	The addressed cache block is valid in this cache, and the data in the block is consistent with system memory. Unlike the exclusive state, however, the block may also simultaneously be in the shared state in another cache in the system. The data in the block may be read at any time by this processor; however, before it may be written with any newer data by this processor, it must first be removed (de-allocated) from all other caches in the system. **Note:** This state is only available when HID2[MESI] is set.
Invalid (I)	This state indicates that the addressed cache block is not resident in the cache.

4.4.2.1 MEI Coherency Protocol

The default cache coherency protocol is a coherent subset of the standard MESI four-state cache protocol that omits the shared state. Since data cannot be shared using MEI, the e300 core signals all cache block fills as if they were write misses (read-with-intent-to-modify, or RWITM), flushing the corresponding copies of the data in all caches external to the core prior to the core cache block fill operation. Following the cache block load, the core is the exclusive owner of the data and may write to it without a bus broadcast transaction.

To maintain this coherency, all global reads observed on the bus by the e300 core are snooped as if they are writes, causing the core to write a modified cache block back to memory and invalidate the cache block, or simply invalidate the cache block if it is unmodified. The exception to this rule occurs when a snooped transaction is a single-beat read (implying caching-inhibited), in which case the core does not invalidate the snooped cache block. If the cache block is modified, the block is written back to memory, and the cache block is marked exclusive. If the cache block is marked exclusive when snooped, no bus action is taken, and the cache block remains in the exclusive state. This treatment of caching-inhibited reads decreases the possibility of data thrashing by allowing noncaching devices to read data without invalidating the entry from the core data cache.

4.4.2.1.1 MEI State Transitions

Figure 4-5 shows the state transitions when the core is configured for MEI coherency protocol (HID2[MESI] = 0). Figure 4-5 assumes that the WIM bits for the page or block are set to 001; that is,

write-back, caching-not-inhibited, and memory-coherency-required. The MEI protocol is the default cache coherency protocol for the e300 core.

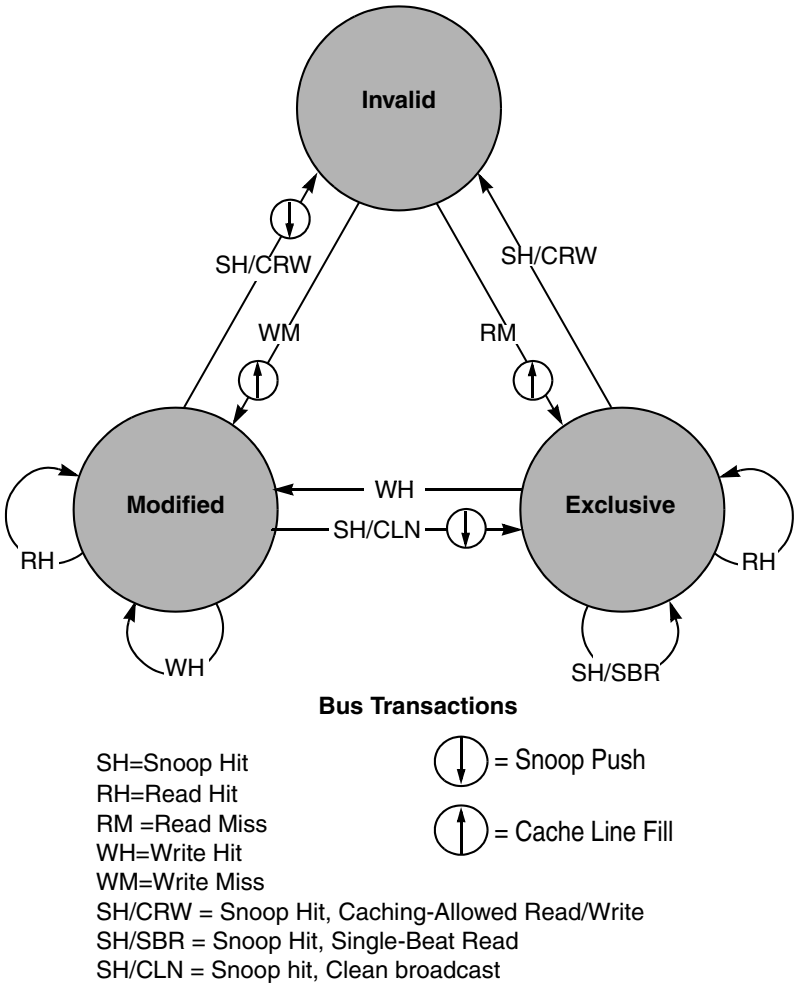


Figure 4-5. MEI Cache Coherency Protocol—State Diagram (WIM = 001)

4.4.2.2 MESI Coherency Protocol

Before enabling MESI mode, the data cache must first be flushed and a flash invalidate performed to empty the cache of any valid data.

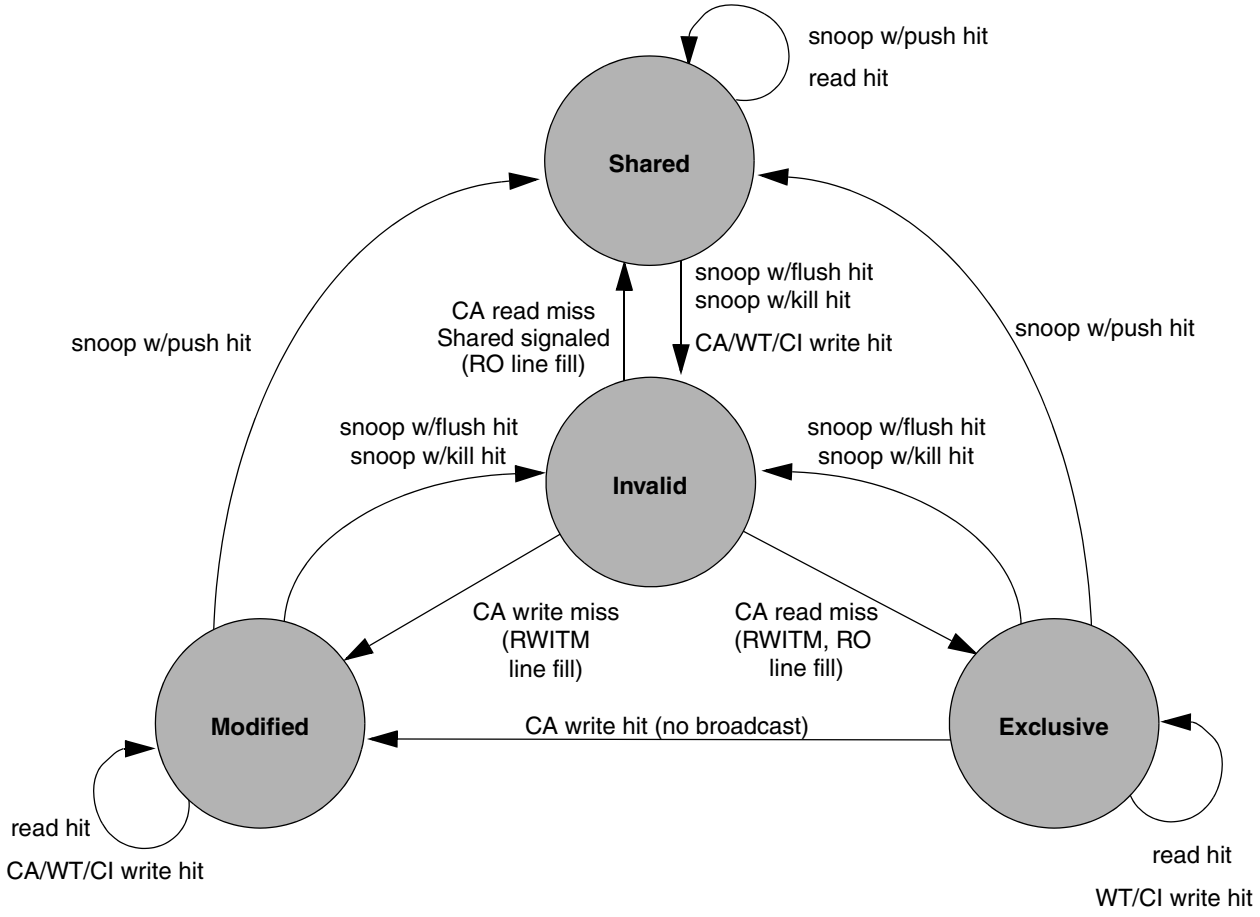
When operating in the four-state protocol, a store-hit to a shared line is treated as a cache miss, and the line is re-fetched from memory with intent-to-modify status on the bus. The newly re-fetched line replaces the same way-entry in the cache. Also, any clean operation in the data cache (**dcbst**, or any snoop-clean type) puts the cache line in the shared state. In MEI mode, a clean operation puts the line in the exclusive state.

When MESI mode is enabled, other nuances of core operation are affected. The transaction type signals on the bus reflect the MESI intention rather than the MEI intention. In particular, a load-miss operation (including for **lwarx**) uses the READ bus transaction, rather than a RWITM bus transaction as is required for the MEI protocol. The **dcbt** instruction causes a READ transaction, and the **dcbtst** instruction causes a RWITM transaction. Note that MMU translation should be enabled when using these instructions. In

addition, instruction fetches are performed as READ transactions, and signal the *glb* pin according to the true WIMG status, including signaling global when the MMU translation is disabled (real addressing mode).

4.4.2.2.1 MESI State Transitions

Figure 4-6 shows the state transitions when the core is configured for MESI coherency protocol (HID2[MESI] = 1). Figure 4-6 assumes that the WIM bits for the page or block are set to 001; that is, write-back, caching-not-inhibited, and memory-coherency-required.



CA = Caching-Allowed, WT= Write-Through, CI = Caching-Inhibited
RO = Read-only
RWITM = Read-with-Intent-to-Modify
snoop w/push = RO bus request, performs a copy-back if the block is in the modified state
snoop w/flush = RWITM bus request, performs a copy-back if the block is in the modified state
snoop w/kill = broadcasts an invalidate on the CSB, no copy-back if the block is in the modified state

Figure 4-6. MESI Cache Coherency Protocol—State Diagram (WIM = 001)

4.4.2.3 Load and Store Coherency Summary

Table 4-4 provides a summary of memory coherency actions performed by the e300 core on load operations. Caching-inhibited cases are not considered in this table.

Table 4-3. Memory Coherency Actions on Load Operations

Cache State	Transaction Generated	External Response	Action
M	None	Don't care	Read from cache
E	None	Don't care	Read from cache
S[1]	None	Don't care	Read from cache
I	READ	No response	Load data and mark exclusive
I	READ	Share signaled[1]	Load data and mark shared[1]
I	READ	Retry signaled	Retry read operation

1 MESI mode only

Table 4-4 provides an overview of memory coherency actions on store operations. This table does not include caching-inhibited or write-through cases.

Table 4-4. Memory Coherency Actions on Store Operations

Cache State	Transaction Generated	External Response	Action
M	None	Don't care	Modify cache
E	None	Don't care	Modify cache and mark modified
S[1]	RWITM	No response	Load data, modify it, and mark as modified
S[1]	RWITM	Retry signaled	Retry the RWITM
I	RWITM	No response	Load data, modify it, and mark as modified
I	RWITM	Retry signaled	Retry the RWITM

1 MESI mode only

The RWITM transactions involve selecting a replacement class and casting-out modified data that may have resided in that replacement class.

4.4.2.4 Coherency in Single-Processor Systems

The following situations concerning coherency can be encountered within a single-processor system:

- Load or store to a caching-inhibited page (WIM = 0bx1x) and a cache hit occurs.

  Caching is inhibited for this page (I = 1)—Load or store operations to a caching-inhibited page that hit in the cache cause boundedly undefined results.

- Store to a page marked write-through (WIM = 0b10x) and a cache read hit to a modified cache block.

  This page is marked as write-through (W = 1)—The core pushes the modified cache block to memory and the block remains marked modified (M).

Note that when WIM bits are changed, it is critical that the cache contents reflect the new WIM bit settings. For example, if a block or page that had allowed caching becomes caching-inhibited, software should ensure that the appropriate cache blocks are flushed to memory and invalidated.

4.4.3 Core-Initiated Load/Store Operations

Load and store operations are assumed to be weakly ordered. In general, the load/store unit (LSU) can perform load operations that occur later in the program ahead of store operations, even when the data cache is disabled. Any load followed by any store is performed in order. See Section 4.4.3.2, “Sequential Consistency of Memory Accesses” for more information.

4.4.3.1 Performed Loads and Stores

The PowerPC architecture defines a performed load operation as one that has the addressed memory location bound to the target register of the load instruction. The architecture defines a performed store operation as one where the stored value is the value that any other processor will receive when executing a load operation (that is, of course, until it is changed again). With respect to the core, caching-allowed (WIMG = *n*0*nn*) loads and caching-allowed write-back (WIMG = 00*nn*) stores are performed when they have arbitrated to address the cache block in the data cache or the coherent system bus (CSB). Loads are considered performed at the data cache only if the cache contains a valid copy of that address. Write-back stores are considered performed at the data cache only if the cache contains a valid copy of that address. Caching-inhibited (WIMG = *n*1*nn*) loads, caching-inhibited (WIMG = *n*1*nn*) stores, and write-through (WIMG = 10*nn*) stores are performed when they have been successfully presented to the CSB.

4.4.3.2 Sequential Consistency of Memory Accesses

The PowerPC architecture requires that all memory operations executed by a single processor be sequentially consistent with respect to that processor. This means that all memory accesses appear to be executed in program order with respect to interrupts and data dependencies.

The e300 load/store unit is very simple, allowing only one load at a time and only one (or one-and-a-half) stores at a time. However, loads are allowed to bypass caching-allowed stores once interrupt checking has been performed for the store, but data dependency checking is handled in the load/store unit so that a load will not bypass a store with an address match. Loads do not bypass caching-inhibited stores.

Although memory accesses that miss in the cache are forwarded to the CSB interface, all potential synchronous interrupts have been resolved before the cache. In addition, although subsequent memory accesses can address the cache, full coherency checking between the cache and the memory queue is provided to avoid dependency conflicts.

4.4.3.3 Enforcing Load/Store Ordering

As defined by the PowerPC architecture, the **eieio** instruction provides an ordering function for the effects of certain classes of load and store instructions executed by a given processor. These classes of instructions are automatically ordered by the LSU on the e300 core, and thus core treats the **eieio** instruction as a no-op.

For the e300 core, caching-inhibited load and store operations are performed in strict program order. Therefore, the caching-inhibited WIMG bit may be used to strictly order memory accesses. In addition, the **sync** instruction may be used to enforce the ordering of any class of load/store operations. The **sync** instruction provides a stronger ordering on the bus by requiring the transaction to complete on the bus. The e300 does not broadcast **eieio** and **sync** instructions.

4.4.3.4 Atomic Memory References

The Load Word and Reserve Indexed (**lwarx**) and Store Word Conditional Indexed (**stwcx.**) instructions provide an atomic update function for a single, aligned word of memory. While an **lwarx** instruction should normally be paired with an **stwcx.** instruction with the same effective address, an **stwcx.** instruction to any address will cancel the reservation. For detailed information on these instructions, refer to Chapter 3, “Instruction Set Model,” in this book and Chapter 8, “Instruction Set,” in the *Programming Environments Manual*.

4.5 Cache Control

The core provides several means of cache control through the use of the memory/cache access attributes (WIMG bits), implementation-specific control bits in the HID0 and HID2 registers, and dedicated cache control instructions. Memory block/page level cache control is provided by the WIMG bits as described in Section 4.4.1, “Memory/Cache Access Attributes (WIMG Bits).” The following sections describe the HID0/HID2 register controls and the cache control instructions.

4.5.1 Cache Control Parameters in HID0 and HID2

The L1 caches are controlled by programming specific bits in the HID0 and HID2 special-purpose registers. This section describes the HID register cache control bits.

4.5.1.1 Cache Parity Error Reporting—HID0[ECPE]

Both instruction and data caches support parity generation, checking, and reporting. Cache parity generation is always on and stored in the cache, but the reporting of cache parity errors is enabled or disabled by using the enable cache parity error bit, HID0[ECPE]. When cache parity errors are enabled, instruction and data cache parity errors are reported by a machine check interrupt.

4.5.1.2 Data Cache Enable—HID0[DCE]

The data cache may be disabled by using the data cache enable bit, HID0[DCE]. DCE is cleared on power-up, disabling the data cache. When the data cache is in the disabled state (HID0[DCE] = 0), the cache tag state bits are ignored, and all accesses are propagated to the CSB as single-beat transactions.

The changing of the DCE bit must be preceded by a **sync** instruction to prevent the cache from being enabled or disabled in the middle of a data access. In addition, the cache must be globally flushed before it is disabled to prevent coherency problems when it is re-enabled.

Although snooping is not performed when the data cache is disabled, cache operations (caused by the **dcbz**, **dcbf**, **dcbst**, and **dcbi** instructions) are not affected by disabling the cache. Thus, there is a risk of coherency errors.

Regardless of the state of DCE, load and store operations are assumed to be weakly ordered. Thus, the LSU can perform load operations that occur later in the program ahead of store operations, even when the data cache is disabled. However, strongly ordered load and store operations can be enforced through the setting of the I bit (of the page WIMG bits) when address translation is enabled. Note that when address translation is disabled, the default WIMG bits cause the I bit to be cleared (accesses are assumed to be caching-allowed); thus, the accesses are weakly ordered. Refer to Section 4.4.1.2, "Caching-Inhibited Attribute (I)," for a description of the operation of the I bit, and Section 6.2, "Real Addressing Mode," for a description of the WIMG bits when address translation is disabled.

4.5.1.3 Data Cache Lock—HID0[DLOCK]

The entire contents of the data cache may be locked by setting the data cache lock bit, HID0[DLOCK]. For a locked data cache, there are no new tag allocations except those caused by **dcbz** instructions. The setting of DLOCK must be preceded by a **sync** instruction to prevent the cache from being locked during a data access.

A locked data cache supplies data normally on a cache hit, but cache misses are treated as caching-inhibited accesses. On a miss, the transaction to the CSB is single-beat; however, *ci* still reflects the state of the I bit in the MMU for that page, regardless of whether the cache is locked or disabled. A snoop hit to a locked data cache performs as if the cache were not locked. Any cache block invalidated by a snoop hit remains invalid until the cache is unlocked.

The e300 core also provides data cache way-locking in addition to entire data cache locking as described in Section 4.5.1.4, "Data Cache Way-lock—HID2[DWLCK]."

4.5.1.4 Data Cache Way-lock—HID2[DWLCK]

Locking only a portion of the data cache is accomplished by locking ways within the cache using HID2[DWLCK]. Locking always begins with the first way (way 0) and is sequential. That is, it is valid to lock ways 0, 1, and 2, but it is not possible to lock just way 0 and way 2. When using way-locking (without DLOCK) at least one way is always left unlocked. The maximum number of lockable ways is seven on e300c1 and three ways on e300c2 and e300c3.

4.5.1.5 Data Cache Flash Invalidate—HID0[DCFI]

The data cache flash invalidate bit, HID0[DCFI], is used to invalidate the entire data cache in a single operation. Note that using DCFI invalidates the cache in a single cycle on e300c1; however, on e300c2 and e300c3, DCFI is a 128-cycle sequential reset of the cache. There is no broadcast of a flash invalidate operation, and any modified data in the cache is lost. Flash invalidation of the data cache is accomplished by setting DCFI and subsequently clearing DCFI in two consecutive **mtspr**[HID0] instructions.

The data cache is automatically invalidated when the core is powered up and during a hard reset. However, a soft reset does not automatically invalidate the data cache. Software must set and clear HID0[DCFI] to invalidate the entire data cache after a soft reset.

4.5.1.6 Instruction Cache Enable—HID0[ICE]

The instruction cache may be disabled through the use of the instruction cache enable bit, HID0[ICE]. ICE is cleared at power-on reset, disabling the instruction cache. To prevent the cache from being enabled or disabled in the middle of an instruction fetch, an **isync** instruction should be issued before changing the value of ICE.

When the instruction cache is in the disabled state, the cache tag state bits are ignored and all accesses are propagated to the bus as single-beat transactions when HID2[IFEB] is cleared; when HID2[IFEB] is set, all accesses are propagated as burst transactions. Note that disabling the instruction cache does not affect the translation logic; translation for instruction accesses is controlled by MSR[IR].

4.5.1.7 Instruction Cache Lock—HID0[ILOCK]

The entire contents of the instruction cache may be locked through the use of HID0[ILOCK]. A locked instruction cache supplies instructions normally on a cache hit, but cache misses are treated as caching-inhibited accesses. On a miss, the transaction to the CSB is single-beat; however, *ci* still reflects the state of the I bit in the MMU for that page, regardless of whether the cache is locked or disabled. The setting of the ILOCK bit must be preceded by an **isync** instruction to prevent the instruction cache from being locked during an instruction access.

The core also provides instruction cache way-locking in addition to entire instruction cache locking, as described in Section 4.10, “Applications Information—Cache Locking.”

4.5.1.8 Instruction Cache Way-Lock—HID2[IWLCK]

Locking only a portion of the instruction cache is accomplished by locking ways within the cache using HID2[IWLCK]. Locking always begins with the first way (way 0) and is sequential. That is, it is valid to lock ways 0, 1, and 2, but it is not possible to lock just way 0 and way 2. Way-locking (without ILOCK) always leaves at least one way unlocked. The maximum number of lockable ways is seven on e300c1 and three ways on e300c2 and e300c3.

4.5.1.9 Instruction Cache Flash Invalidate—HID0[ICFI]

The instruction cache flash invalidate bit, HID0[ICFI], is used to invalidate the entire instruction cache in a single operation. Note that using ICFI invalidates the cache in a single cycle on e300c; however, on e300c2 and e300c3, ICFI is a 128-cycle sequential reset of the cache. Flash invalidation of the instruction cache is accomplished by setting ICFI and subsequently clearing ICFI in two consecutive **mtspr**[HID0] instructions.

The instruction cache is automatically invalidated when the core is powered up and during a hard reset. However, a soft reset does not automatically invalidate the instruction cache. Software must set and clear HID0[ICFI] to invalidate the entire instruction cache after a soft reset.

4.5.1.10 Instruction Cache Way Protect—HID2[ICWP]

Typically, instruction cache management routines rely on **icbi** instructions or flash invalidate (HID0[ICFI]) operations to clear out part or all of the instruction cache for program-related operations, such as for process changes or MMU page deallocation. Often, these cache management operations clear out both the locked and unlocked portions of the instruction cache (**icbi** invalidates all ways of a cache set unconditionally, and HID0[ICFI] invalidates the entire instruction cache unconditionally).

The e300 core has a new instruction cache way protection feature. By setting HID2[ICWP] = 1, any locked ways in the instruction cache are protected from cache block (**icbi**) or flash (HID0[ICFI]) invalidation. This allows any locked portion of the instruction cache to have the same persistence as main memory, while still allowing the remaining unlocked portions of the instruction cache to be managed by the program.

4.5.1.11 Cache Operation Broadcasting—HID0[ABE]

In MEI mode, the core does not broadcast cache operations caused by cache instructions. They are intended for the management of the local cache but not for other caches in the system. The HID0[ABE] bit enables the cache management instructions to unconditionally cause a bus transaction (either a push as a result of modified data that needs to be written back to memory or an address-only transaction). The use of ABE is generally intended to extend cache management to a front-side level-2 cache. In MESI mode, the cache management instructions are automatically broadcast on the bus as defined by the M-bit of the WIMG field.

4.5.2 Cache Control Instructions

The PowerPC architecture defines instructions for controlling both the instruction and data caches when they exist. The core interprets the cache control instructions (**dcbt**, **dcbtst**, **dcbz**, **dcbst**, **dcbf**, **dcbi**, **icbt**, and **icbi**) as if they pertain only to the core caches. They are not intended for use in managing other caches in the system.

The **dcbz** instruction causes an address-only broadcast on the bus if the addressed cache block is marked memory-coherency-required (global) through the WIMG bits. This broadcast is performed for coherency reasons; the **dcbz** instruction is effectively a normal store class instruction.

In MEI mode, the execution of a **dcbf**, **dcbi**, or **dcbst** instruction causes an address-only broadcast if the HID0[ABE] bit is set. Also, in MEI mode, the **dcbz** instruction is the only cache operation that is snooped by the core.

In MESI mode, the execution of the **dcbf**, **dcbi**, and **dcbst** instructions are broadcast if the HID0[ABE] bit is set or if the addressed cache block is marked memory-coherency-required. Also in MESI mode, the core snoops the **dcbst**, **dcbz**, **dcbf**, and **dcbi** instructions.

The ability of the core to optionally perform address-only broadcasts when executing the **dcbi**, **dcbf**, and **dcbst** instructions allows for managing coherency of external caches if they are present.

4.5.2.1 Data Cache Block Touch (dcbt) Instruction

This instruction provides a method for improving performance through the use of software-initiated prefetch hints. The core performs the fetch when the address hits in the TLB or BAT registers, and when it is a permitted load access from the addressed page. The operation is treated similarly to a byte load operation with respect to coherency.

The **dcbt** instruction which misses in the data cache is treated as a burst read operation on the bus, with a transaction type of RWITM if operating in three-state MEI mode, or READ if operating in four-state MESI mode. The resulting cache line is marked as exclusive if operating in three-state MEI mode. If operating in four-state MESI mode, it is marked as exclusive or shared, depending on the shared indication on the CSB.

The **dcbt** instruction is treated as a no-op if any of the following apply:

- It hits in the data cache.
- The target address is mapped to caching-inhibited (even if the data cache is disabled).
- Touch load operations are disabled by HID0[NOPTI].
- The target address is mapped as guarded.
- The address translation does not hit in the TLB or BAT mechanism.
- The address translation does not have load access permission.

4.5.2.2 Data Cache Block Touch For Store (dcbtst) Instruction

The **dcbtst** instruction, like the data cache block touch instruction (**dcbt**), allows software to prefetch a cache block in anticipation of a store operation (RWITM).

The **dcbtst** instruction is treated similarly to the **dcbt** instruction, except that the transaction type RWITM is always used if operating in 4-state MESI mode, and the resulting cache line is always marked as exclusive (not modified). When operating in MEI mode, the **dcbtst** has exactly the same behavior as **dcbt**.

4.5.2.3 Data Cache Block Clear To Zero (dcbz) Instruction

If the block containing the byte addressed by the EA is in the data cache, all bytes are cleared.

If memory coherency is required (WIMG = *nn*1*n*), the **dcbz** instruction broadcasts on the bus on a cache miss or hit to a shared block. It is the only cache control instruction that may broadcast without being specifically enabled using HID0[ABE]. The **dcbz** instruction is essentially a store operation rather than a cache management operation and, therefore, always participates in normal cache coherency as directed by the M-bit.

The core initiates an alignment interrupt when the operand of a **dcbz** is in a page that is write-through or caching-inhibited or when the data cache is disabled. The **dcbz** instruction is not executed in these cases.

If the **dcbz** instruction is otherwise not aborted, it allocates into the cache and performs an address broadcast, if necessary, for normal store coherency. The resulting cache line is marked as modified. In the case where the **dcbz** operation hits in the cache, it re-allocates to that cache line. Note that the **dcbz** instruction ignores HID0[DLOCK] setting and always allocates a tag. That is, when DLOCK is set, the **dcbz** occurs as if the cache were not locked.

4.5.2.4 Data Cache Block Store (dcbst) Instruction

This instruction is treated as a load to the addressed byte with respect to address translation and protection.

If the address hits in the data cache, the target cache line is pushed to memory as a burst write bus transaction. The cache line remains in the data cache and is marked as exclusive if operating in MEI mode, or marked as shared if operating in MESI mode. If no push is required from the data cache, the **dcbst** instruction instead conditionally generates an address broadcast operation on the bus. The address broadcast is contingent upon whether HID0[ABE] is set, or whether the target address is mapped as memory-coherency-required (M = 1) in MESI mode.

The **dcbst** instruction is always effective on the data cache or the bus, regardless of WIMG settings or whether the data cache is enabled.

4.5.2.5 Data Cache Block Flush (dcbf) Instruction

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. This instruction is treated as a load to the addressed byte with respect to address translation and protection.

If the address hits in the cache, and the block is in the modified state, the modified block is written back to memory and the cache block is invalidated. If the address hits in the cache, and the cache block is in the exclusive or shared state, the cache block is invalidated. If the address misses in the cache, no action is taken.

The function of this instruction is independent of the WIMG bit settings of the block or PTE containing the effective address. However, the execution of **dcbf** broadcasts an address-only flush transaction on the CSB if HID0[ABE] is set or if operating in four-state MESI mode and the target address is marked memory-coherency-required. Execution of a **dcbf** instruction affects the cache even if the cache is disabled.

4.5.2.6 Data Cache Block Invalidate (dcbi) Instruction

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. This instruction is treated as a store to the addressed byte with respect to address translation and protection.

The **dcbi** instruction should be used with caution on the e300 core.

If the address hits in the cache, the cache block is invalidated, regardless of the state of the cache block. Even if the cache block is modified, it is not pushed to main memory and the associated data is lost. Because this instruction may effectively destroy modified data, it is privileged (that is, **dcbi** is available to programs at the supervisor privilege level, MSR[PR] = 0). A BAT or TLB protection violation for a **dcbi** translation generates a DSI interrupt.

The function of this instruction is independent of the WIMG bit settings of the block or PTE containing the effective address. However, the execution of **dcbi** broadcasts an address-only kill transaction on the CSB if HID0[ABE] is set or if operating in four-state MESI mode and the target address is marked memory-coherency-required. Execution of a **dcbi** instruction affects the cache even if the cache is disabled.

4.5.2.7 Instruction Cache Block Touch (icbt) Instruction

The **icbt** instruction performs a bus read operation from the bus and allocates into the instruction cache. This instruction is new to the e300 core, and supplements the instruction cache locking mechanisms and the new way-protect feature.

The **icbt** instruction is treated as a no-op if touch load operation is disabled by HID0[NOPTI].

The **icbt** instruction is effective, regardless of WIMG settings, instruction or data cache enable status, and the instruction cache lock status (that is, unconditional allocate). This allows the **icbt** instruction to easily initialize the locked portion of the instruction cache before enabling.

Note that to prevent user-level code from inadvertently overwriting a supervisor-level page that has been locked in the instruction cache, that page of memory should be protected with appropriate MMU translation and access privileges, or by using HID0[NOPTI] to treat the **icbt** instruction as a no-op.

The **icbt** instruction is dispatched to the load/store unit and data cache, and, therefore, goes through data-side address translation. It is treated similarly to **dcbt** for translation and storage protection purposes. The data cache then issues the **icbt** operation to the bus unit without checking for a data cache hit. Therefore, if the instruction block is already residing in the data cache (for example, due to self-modifying code), the program must first perform a **dcbf** of that address to flush that data block back to main memory for the subsequent **icbt**.

A **sync** instruction must follow an **icbt** instruction to ensure all cache load operations are completed. An **isync** instruction must also follow an **icbt** if the newly touched and loaded instructions is expected to be fetched immediately after the **icbt** is executed.

4.5.2.8 Instruction Cache Block Invalidate (icbi) Instruction

The **icbi** instruction unconditionally invalidates all ways of the target cache set (that is, **icbi** invalidates by tag index only). No address comparison is performed to check for a cache hit. The **icbi** instruction does not broadcast to the bus.

If the instruction cache way-protect bit, HID2[ICWP], is set, only the non-locked ways of the instruction cache set are invalidated by the **icbi** instruction.

An **icbi** instruction should always be followed by a **sync** and an **isync** instruction. This ensures that the effects of the **icbi** are seen by the instruction fetches following the **icbi** itself. For self-modifying code, the following sequence should be used to synchronize the instruction stream:

1. **dcbst** (push new code from the data cache out to memory)
2. **sync** (wait for the **dcbst** to complete)
3. **icbi** (invalidate the old instruction cache entry in this processor)
4. **sync** (wait for the **icbi** to complete its bus operation)
5. **isync** (re-sync this processor’s instruction fetch)

The second **sync** instruction ensures completion of all prior **icbi** instructions. Note that the second **sync** instruction is not shown in Section 5.1.5.2, “Instruction Cache Instructions,” in *The Programming Environments Manual*.

Since the **sync** instruction strongly serializes the memory subsystem, performance of code containing several **icbi** instructions can be improved by batching the **icbi** instructions together such that only one **sync** instruction is used to synchronize all the **icbi** instructions in the batch.

4.6 Cache Operations

This section describes the three types of operations that can occur to the caches, and how these operations are implemented in the e300 core.

4.6.1 Data Cache Fill Operations

A cache block fill is caused by a cacheable load or store miss in the cache. The cache block that corresponds to the missed address is updated by a burst transfer of the data from system memory. If a read miss occurs in a system with multiple bus masters, and the data is modified in another cache, the modified data is first written to external memory before the cache fill occurs.

When the core is configured with a 64-bit data bus, cache blocks are loaded in four beats of 8 bytes each. When the core is configured with a 32-bit bus, cache block loads are performed with eight beats of 4 bytes each. The burst load is performed as critical-double-word-first. The data cache is blocked to subsequent load/store operations until the load completes. The critical-double-word is simultaneously written to the cache and forwarded to the requesting unit, thus minimizing stalls due to load delays.

4.6.2 Instruction Cache Fill Operations

When the core is configured with a 64-bit data bus, instruction cache blocks are loaded in four beats of 8 bytes each. When the core is configured with a 32-bit bus, cache block loads are performed with eight beats of 4 bytes each. The burst fetch is performed as critical-double-word-first. On a cache miss, the critical and following double words fetched from memory are simultaneously written to the instruction cache and forwarded to the dispatch queue, thus minimizing stalls due to cache fill latency. The instruction cache allows sequential fetching from the remainder of the cache block during a cache block load.

4.6.3 Instruction Fetch Cancel Extension

In superscalar architectures, instructions are routinely fetched ahead of the time they are needed, but these prefetched instructions may be cancelled due to instruction redirection, such as by branches or interrupts. The instruction fetch cancel extension improves the utilization of the instruction cache during such cancel operations.

The instruction fetch cancel extension allows a new instruction fetch to be issued to the cache or to the bus if a cancelled instruction fetch is pending or active on the bus. This supports hit-under-cancel and miss-under-cancel instruction fetch operations. This feature is enabled using HID2[IFEC].

4.6.4 Data Cache Cast-Out Operations

The core uses a PLRU replacement algorithm to determine which of the possible cache locations should be used for a cache update on a cache miss. Adding a new block to the cache causes any modified data

associated with the replacement element to be written back, or cast out, to system memory to maintain memory coherence. Refer to Section 4.6.7, “Cache Block Replacement Selection,” for more information on how the replacement block is selected.

4.6.5 Cache Block Push Operation

When a cache block in the core is snooped and hit by another bus master and the data is modified, the cache block must be written to memory and made available to the snooping device. The cache block that is hit, is pushed out onto the CSB. If the snooped transaction is a write-with-kill transaction that hits a modified cache block, no push will be performed.

There is no snooping of the instruction cache.

4.6.6 Data Cache Queue Sharing Extension

In previous G2 cores, two write queues are available for data cache burst write operations: one reserved for cache replacements only, and one for snoop pushes only. (There is also a third write queue always available for non-burst write operations.)

The e300 features a data cache queue sharing extension that allows the two burst write queues in the bus unit to be used interchangeably, or shared, for cache replacements and snoop pushes. This allows the data cache to support two outstanding cache replacements or two outstanding snoop push operations on the bus at any given time. This supports greater bus throughput between the data cache and memory, and less stall latency during data cache miss operations due to unavailable bus queues. However, when queue sharing is enabled, the snoop queue may not be available because there may be 2 pending cast-outs. In this case, there may be a window-of-opportunity push from an address other than the snooped address.

Queue-sharing is enabled by setting HID2[EBQS].

4.6.7 Cache Block Replacement Selection

The instruction and data caches both use a three-step selection process to determine which cache way will be used for the instruction or data. First, the core determines if there is a hit to a valid way. If there is a hit, then that way is selected. Next, the core checks to see if there are any invalid ways in the set and chooses the lowest-order, invalid way as the recipient. Last, when there is not a hit, but all eight ways in the set are valid (and not locked), the PLRU algorithm is used to select the replacement target.

For e300c1, there are seven PLRU bits, B[0–6] for each set in the cache. A way is selected for replacement according to the PLRU bit encodings shown in Table 4-5.

Table 4-5. e300c1 PLRU Replacement Way Selection

If the PLRU bits are:						Then the way selected for replacement is:
B0	0	B1	0	B3	0	w0
	0		0		1	w1
	0		1	B4	0	w2
	0		1		1	w3
	1	B2	0	B5	0	w4
	1		0		1	w5
	1		1	B6	0	w6
	1		1		1	w7

For e300c2 and e300c3, the B0 bit is always 0, so there are effectively only three PLRU bits (B1, B3, and B4) for each set in the cache. A way is selected for replacement according to the PLRU bit encodings shown in Table 4-6.

Table 4-6. e300c2 PLRU Replacement Way Selection

If the PLRU bits are:				Then the way selected for replacement is:
B1	0	B3	0	w0
	0		1	w1
	1	B4	0	w2
	1		1	w3

The PLRU algorithm is shown graphically in Figure 4-7.

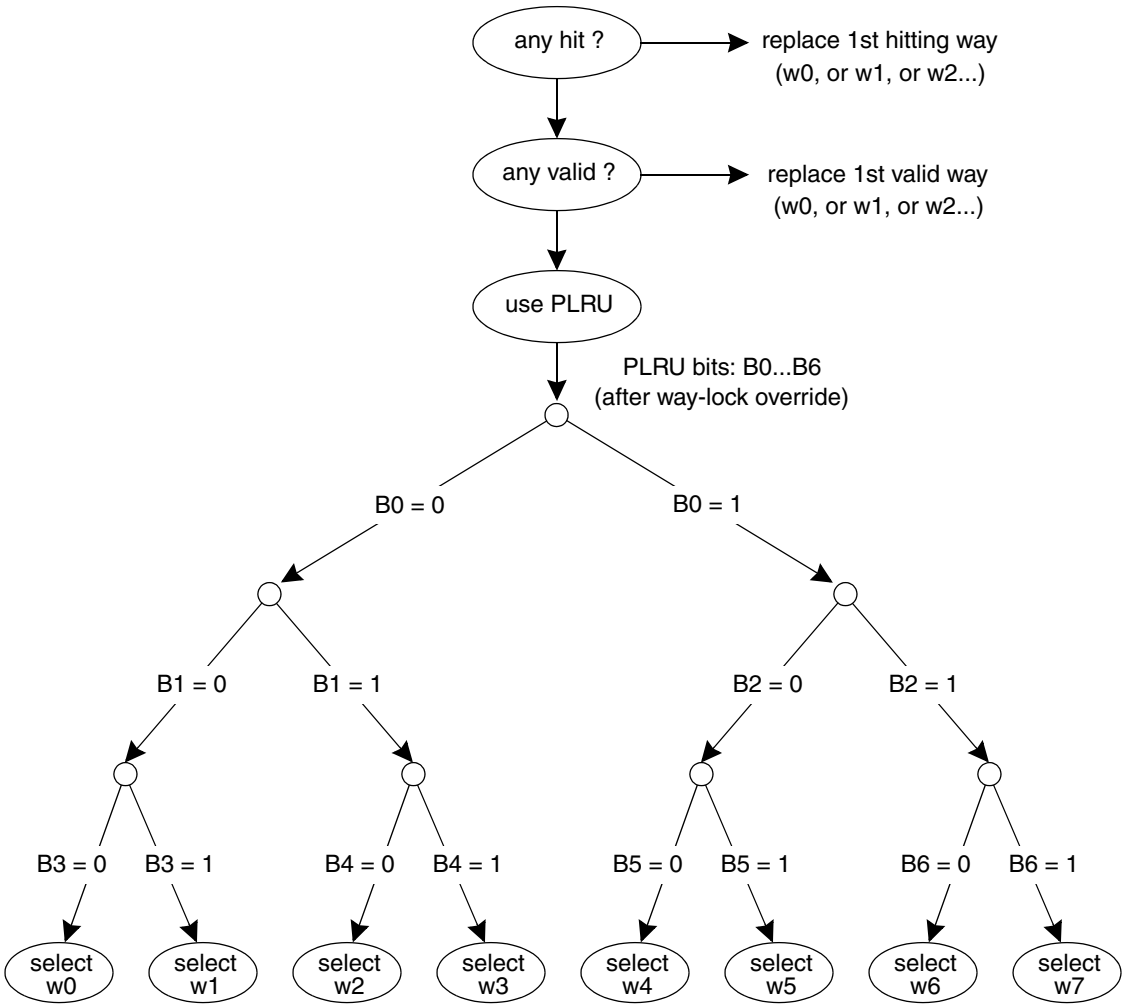


Figure 4-7. PLRU Replacement Algorithm

Note: B0 = 0 always taken on e300c2.

During power-up or hard reset, all the valid bits of the ways are cleared and the PLRU bits are cleared to point to way 0 of each set. This is also the state of the data or instruction cache after setting their respective flash invalidate bits (HID0[DCFI] or HID0[ICFI]).

Each time a cache block is accessed, it is tagged as the most recently used way of the set. For every hit in the cache or when a new block is reloaded, the PLRU bits for the set are updated using the rules specified in Table 4-7.

Table 4-7. PLRU Bit Update Rules

If the current access is to:	Then the PLRU bits in the set are changed to:						
	B0[1]	B1	B2	B3	B4	B5	B6
w0	1	1	x	1	x	x	x
w1	1	1	x	0	x	x	x
w2	1	0	x	x	1	x	x
w3	1	0	x	x	0	x	x
w4	0	x	1	x	x	1	x
w5	0	x	1	x	x	0	x
w6	0	x	0	x	x	x	1
w7	0	x	0	x	x	x	0

x = Does not change

[1] Note that the e300c2 only has 4 ways, so B0 is always 0.

Note that for the e300c1 only three PLRU bits are updated for any given access and for the e300c2 and e300c3 only two PLRU bits are updated for any given access.

In the case of way-locking, the PLRU value read from the cache is first modified before using it to ensure that a locked way is not selected. Because way-locking involves locking an incrementing range of ways starting with way 0 (way 0, or way 0–1, or way 0–2, etc.), the appropriate bits of the PLRU value are simply overridden to 1 in an incrementing fashion away from the locked ways to prevent that range of ways from being selected. In the binary tree figure, this can be visually described as forcing the bits required to reach the locked ways to the value 1 so that the binary tree is traversed down one of the remaining branches.

The final PLRU value for the selected way of the cache set is written back to the cache for load hits, store hits (including **dcbz**), and normal cache allocations. The value written back is adjusted to ensure that way will not be immediately selected the next time(s) a replacement is required for that particular cache set. The PLRU value written back is the PLRU value pointing to that way with its three critical bits inverted. The three critical bits are the three bits that were traversed down the binary tree to reach the final way selection. The remaining bits are left unchanged. Inverting the three critical bits essentially converts the PLRU value to a PMRU (pseudo most-recently-used) value. In the case of way-locking, the way-locking override described above is not factored into the PLRU write-back value. In the case of a cache hit, the PLRU write-back value has the three critical bits inverted for the specific hitting way.

4.7 L1 Cache Parity

The e300 core includes parity checking for both the instruction and data caches. A machine check interrupt is taken upon detecting an instruction or data cache parity error when HID0[ECPE] and MSR[ME] are

both set. For the instruction cache, parity is checked for instruction fetches that hit in the cache. For the data cache, parity is checked for loads that hit in the cache as well as for any cache line writes to memory (replacement copy-backs, **dcbf**/**dcbst** pushes, and snoop copy-backs). Note that data and instruction cache parity generation and checking are always on; the ECPE parameter simply enables reporting of parity errors.

The state of SRR1 reflects the additional machine check conditions of instruction cache parity error (bit 10) or data cache parity error (bit 11). Cache parity errors are logged as non-recoverable. Refer to Section 5.5.2, "Machine Check Interrupt (0x00200)," for more information.

4.8 Bus Interface

The bus interface buffers receive requests from the instruction and data caches, and execute the requests per the CSB protocol. They include address register queues, prioritization logic, and bus control logic. The bus interface also captures snoop addresses for snooping in the cache and address register queues, snoops for reservations, and holds the touch load address for the cache. All data storage for the address register buffers (load and store data buffers) is located in the cache logic. The data buffers are considered temporary storage for the cache and not part of the bus interface.

The general functions and features of the bus interface are as follows:

- Address register buffers that include:
  - Instruction cache load address buffer
  - Data cache load address buffer
  - Data cache cast-out/store address buffers (associated data line buffer located in cache)
  - Data cache single-beat write address buffers (associated data line buffer located in cache)
  - Data cache snoop copy-back address buffer (associated data line buffer located in cache)
  - Reservation address buffer for snoop monitoring
- Pipeline collision detection for data cache buffers
- Reservation address snooping for **lwarx**/**stwcx.** instructions
- One-level or one-and-a-half-level address pipelining
- Load-ahead-of-store capability

Figure 4-8 is a conceptual block diagram of the bus interface. The address register queues hold transaction requests that the bus interface may issue on the CSB independently of the other requests. The bus interface may have up to two transactions operating at any given time through the use of address pipelining.

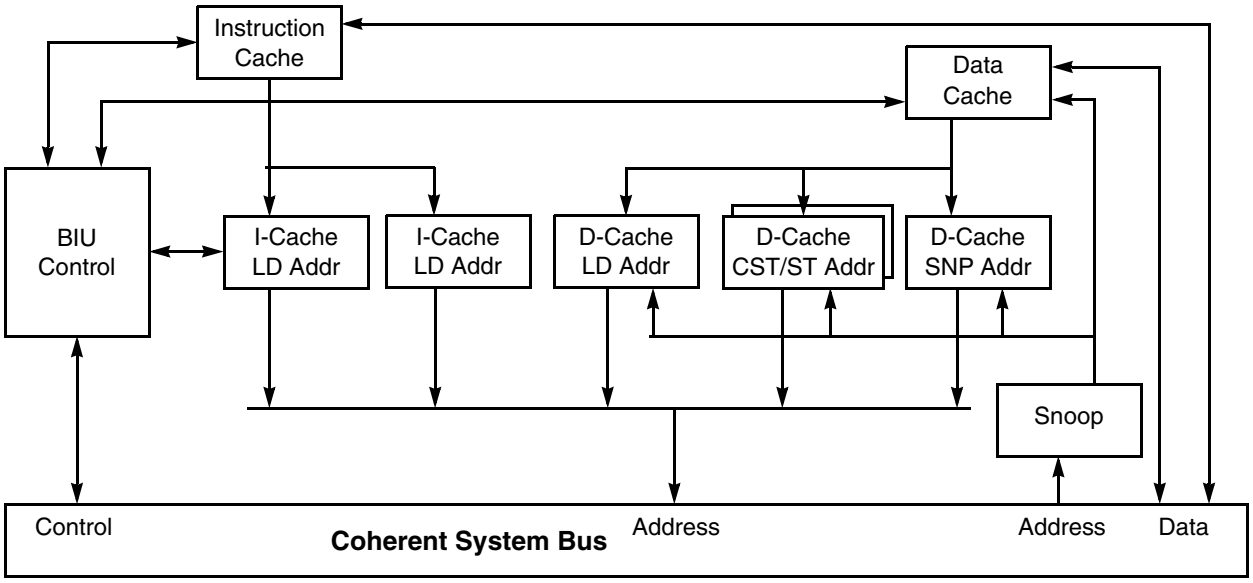


Figure 4-8. Bus Interface Address Buffers

4.9 Caches and CSB Transactions

The core transfers data to and from the data cache in single-beat transactions of two words, or in four-beat transactions of eight words which fill a cache block.

4.9.1 Single-Beat Transactions

Single-beat bus transactions can transfer from 1 to 8 bytes to or from the core. Single-beat transactions can be caused by cache write-through accesses, caching-inhibited accesses (I bit of the WIMG bits for the page is set), accesses that miss when the cache is locked (HID0[DLOCK] is set), accesses when the cache is disabled (HID0[DCE] bit is cleared), and misaligned accesses.

4.9.2 Burst Transactions

Burst transactions on the core always transfer eight words of data at a time and are aligned to a double-word boundary. Burst transactions have an assumed address order. For caching-allowed read operations or caching-allowed, non-write-through write operations that miss the cache, the core presents the double-word aligned address associated with the load or store instruction that initiated the transaction.

As shown in Figure 4-9, this quad word contains the address of the load or store that missed the cache. This minimizes latency by allowing the critical code or data to be forwarded to the processor before the rest of the block is filled. For all other burst operations, however, the entire block is transferred in order (eight-word aligned). Critical-double-word-first fetching on a cache miss applies to both the data and instruction cache.

e300 Core Cache Address Bits 27:28

0 0	0 1	1 0	1 1
A	B	C	D

If the address requested is in double-word A, the address placed on the bus is that of double-word A, and the four data beats are ordered in the following manner:

Beat

0	1	2	3
A	B	C	D

If the address requested is in double-word C, the address placed on the bus will be that of double-word C, and the four data beats are ordered in the following manner:

Beat

0	1	2	3
C	D	A	B

Figure 4-9. Double-Word Address Ordering—Critical-Double-Word-First

4.9.3 Instruction Fetch Burst Enable for Caching-Inhibited Space

In the G2 core, all instruction fetches to caching-inhibited (CI) space results in single-beat bus transactions and in the fetched instructions not being cached. The *ci_b* signal reflects the setting of the I bit for the block or page that contains the address of the current transaction.

In the e300 core, the instruction fetch burst enable extension allows all instruction fetches from caching-inhibited instruction space to be performed on the CSB as burst transactions, similar to caching-allowed instruction space, even though the instructions are not cached. This allows for greater performance to instruction space that is not desired to be cached. With this performance extension, up to an entire cache line may be returned (up to 8 instructions) with one bus operation. The remaining instructions in the burst transaction that follow the critical instruction are forwarded to the instruction fetch unit as they are normally requested, until an instruction redirect or instruction stall occurs. Because instruction fetching does not normally wrap within the cache line, an instruction fetch initiated to double word address 3 of a 32-byte block is still performed as a single-beat read on the CSB. This feature is enabled by setting the HID2[IFEB] register bit.

When the HID2[IFEB] register bit is set and the HID0[ICE] register bit is cleared (instruction cache disabled), any instruction fetch results in burst transactions on the bus, similar to caching-allowed instruction space, but the instruction is not cached.

This feature should not be enabled for systems that do not support bursting from all caching-inhibited instruction space which could otherwise be accessed while burst mode is enabled. This feature affects only caching-inhibited instruction fetches, not caching-inhibited load or store operations.

4.9.4 CSB Operations Caused by Cache Control Instructions

Table 4-8 provides an overview of the bus operations initiated by cache control instructions. The cache control, TLB management, and synchronization instructions supported by the core may affect or be affected by the operation of the CSB. None of the instructions actively broadcast through address-only transactions on the bus (except for **dcbz**), and no broadcasts by other masters are snooped by the core (except for kills and those required by the MESI protocol). The operation of the instructions, however, may indirectly cause bus transactions to be performed, or their completion may be linked to the bus. Table 4-8 summarizes how these instructions may operate with respect to the CSB.

Table 4-8. e300 Bus Operations Caused by Cache Control Instructions

Operation	Cache State	Next Cache State	CSB Operation	Comment
sync	Don't care	No change	None	Waits for data path related queues to complete bus activity
icbi	Don't care	I	None	—
dcbi	Don't care	I	None or Kill block	Broadcast dependent on whether HID0[ABE] is set in MEI mode
dcbf	M	I	Write-with-kill	Block is pushed
dcbf	E, S, I	I	None or Write	Broadcast dependent on whether HID0[ABE] is set in MEI mode
dcbst	M	E (MEI -mode) S (MESI-mode)	Write	Block is pushed
dcbst	E, S, I	No change	None	—
dcbz	E	M	None	Clear all bytes in the block
dcbz	S	M	Kill block	Invalidate cache line Allocate the tag Clear all bytes in the block
dcbz	I	M	Kill block	—
dcbt	M, E	No change	None	—
dcbt	I	No change	Read-with-intent-to-modify (MEI) Read (MESI)	Load the block into cache
dcbtst	I	No change	Read-with-intent-to-modify	Load the block into cache
dcbtst	E, M	No change	None	—

Table 4-8 assumes that the WIM bits are set to 001; that is, since the cache is operating in write-back mode, caching is permitted and coherency is enforced. Table 4-8 does not include caching-inhibited or write-through cases, nor does it completely describe the mechanisms for the operations described.

4.9.5 Snooping

The core maintains data cache coherency in hardware by coordinating activity between the data cache, memory system, and bus interface logic.

The global ($\overline{gbl}$) signal, asserted as part of the address attributes during a bus transaction, enables the snooping hardware of the core. Address bus masters assert $\overline{gbl}$ to indicate that the current transaction is a global access (that is, an access to memory shared by more than one device). If $\overline{gbl}$ is not asserted for the transaction, that transaction is not snooped by the core. The $\overline{gbl}$ signal is not asserted for instruction fetches (except when HID0[IFEM] is set and the instruction address is marked memory-coherency-required); $\overline{gbl}$ is asserted for all data read or write operations when using real addressing mode.

Normally, $\overline{gbl}$ reflects the M-bit value specified for the memory reference in the corresponding MMU translation descriptor(s). Care must be taken to minimize the number of pages marked as global, because the retry protocol enforces coherency and this incurs additional overhead.

As global bus transactions are performed on the bus by other bus masters, the core bus snooping logic monitors the addresses that are referenced. These addresses are compared with the cache tags. If there is a snoop hit, the core bus snooping logic responds with the appropriate snoop status (for example, a retry). Additional snoop action may be forwarded to the cache as a result of a snoop hit in some cases (a cache push of modified data or cache block invalidation). The specific response depends on the transaction type. There are several bus transaction types defined for the CSB. The core snoops these transactions and performs the appropriate action to maintain memory coherency as described in Table 4-9.

Table 4-9. Snoop Response to CSB Transactions

Snooped Transaction Type	e300 Core Response
Clean block	MEI: No action is taken MESI: The clean block operation is an address-only bus transaction initiated when a **dcbst** instruction is executed.The core may have the following response: • If the addressed block is in the invalid or shared state, no action is taken and the state of the addressed block is unchanged. • If the addressed block is in the exclusive state, the address snoop forces the state of the addressed block to shared. • If the addressed block is in the modified state, the address snoop signals retry and initiate a push of the modified block out of the cache and changes the state of the block to shared.
Flush block	MEI: No action is taken MESI: The flush block operation is an address-only bus transaction that initiates when a **dcbf** instruction is executed. The core may have the following response: • If the addressed block is in the invalid state, no action is taken. • If the addressed block is in the exclusive or shared state, the address snoop forces the state of the addressed block to invalid. • If the addressed block is in the modified state, the address snoop signals retry and initiates a flush of the modified block out of the cache and changes the state of the block to invalid. • Any associated reservation is canceled.
Write-with-flush Write-with-flush-atomic	MEI/MESI: Write-with-flush and write-with-flush-atomic operations occur after the processor issues a store or **stwcx.** instruction, respectively. • If the addressed block is in the invalid state, no action is taken. • If the addressed block is in the shared state, the address snoop forces the state of the addressed block to invalid. • If the addressed block is in the exclusive state, the address snoop forces the state of the addressed block to invalid. • If the addressed block is in the modified state, the address snoop signals retry initiate a push of the modified block out of the cache, and change the state of the block to invalid. • Any associated reservation is canceled.

Table 4-9. Snoop Response to CSB Transactions (continued)

Snooped Transaction Type	e300 Core Response
Kill block	MEI/MESI: The kill block operation is an address-only bus transaction initiated when a **dcbz** instruction is executed. When snooped by the core, the addressed cache block is invalidated if in the exclusive or shared state, or flushed to memory and invalidated if in the modified state; and any associated reservation is canceled.
Write-with-kill	MEI/MESI: In a write-with-kill operation, the core snoops the cache for a copy of the addressed block. If one is found, an additional snoop action is initiated internally and the cache block is forced to the invalid state, killing modified data that may have been in the block. Any reservation associated with the block is also canceled. There is an exception to this rule, if there is any pending single-beat-write atomic write-through (**stwcx.**) in BIU, the core treats the write-with-kill as if it were a write-with-flush bus transaction.
Read Read-atomic	MEI: The read operation is used by most single-beat-read and burst read operations on the bus. All burst reads observed on the bus are snooped as if they were writes (RWITM), causing the addressed cache block to be flushed. A read on the bus with the $\overline{gbl}$ signal asserted causes the following responses: • If the addressed block in the cache is in the invalid state, the core takes no action. • If the addressed block in the cache is in the exclusive state, the block is invalidated. • If the addressed block in the cache is in the modified state, the block is flushed to memory and the block is invalidated. • If the snooped transaction is a single-beat-read (caching-inhibited) and the block in the cache is in the exclusive state, the snoop causes no bus activity and the block remains in the exclusive state. If the block is in the cache in the modified state, the core initiates a push of the modified block out to memory and marks the cache block as exclusive. Read-atomic operations appear on the bus in response to **lwarx** instructions and generate the same snooping responses as read operations. MESI: The read operation is used by most single-beat-read and burst read operations on the bus. In the MESI protocol, burst reads are treated as reads. All burst reads observed on the bus are snooped as if they were reads. A read on the bus with the $\overline{gbl}$ signal asserted causes the following responses: • If the addressed block in the cache is in the invalid state, the core takes no action. • If the addressed block in the cache is in the exclusive state, the core signals shared and the address snoop forces the state of the addressed block to shared. • If the addressed block in the cache is in the modified state, the address snoop signals retry and shared and initiates a push of the modified block out of the cache and changes the state of the block to shared. Read-atomic operations appear on the bus in response to **lwarx** instructions and generate the same snooping responses as read operations.
Read-with-intent-to-modify (RWITM) RWITM-atomic	MEI/MESI: A RWITM operation is issued to acquire exclusive use of a memory location for the purpose of modifying it. • If the addressed block is in the invalid state, the core takes no action. • If the addressed block in the cache is in the exclusive or shared state, the core initiates an additional snoop action to change the state of the cache block to invalid. • If the addressed block in the cache is in the modified state, the block is flushed to memory and the block is invalidated. • Any associated reservation is canceled (MESI state only). The RWITM-atomic operations appear on the bus in response to **stwcx.** instructions and are snooped like RWITM instructions.
sync	No action is taken
TLB invalidate	No action is taken

In addition to the retries described in Table 4-9, the core may signal retry for any bus transaction due to internal conflicts that prevent the appropriate snooping. The following scenarios cause the core to signal a retry:

- Snoop hits to a block in the M state (flush or clean)

  This case is a normal snoop hit and results in retry being signaled if the snooped transaction is a flush or clean request. If the snooped transaction is a kill request, retry is not signaled.
- Snoop hits to line in the cast-out buffer. The cast-out buffer is kept coherent with main memory, and snoop operations that hit in the cast-out buffer cause retry to be signaled.
- Snoop attempt during the tag allocation period from **dcbz** instruction or load or store operations. During the execution of a **dcbz** instruction or during a load or store operation that requires a cache line cast-out, the cache tags are inaccessible during the first and last cycle of the operation. The period of any associated cast-out due to reallocation overlaps the tag allocate period.
- Snoop attempt during the cycle when a **dcbf**, **dcbst**, or **dcbi** instruction is updating the tag. If the EA of a **dcbf** or **dcbst** instruction hits in the cache, the tag will be changed to its new state. During that clock, the tag is not accessible and snoop transactions during that cycle cause retry to be signaled.
- Snoop hits in the data load buffer for a burst read while the data is still being transferred from memory (that is, data is in transit).
- Other internal resource collisions

While the e300 core provides the hardware required to monitor bus traffic for coherency, the core data cache tags are single-ported, and a simultaneous load or store and snoop access represent a resource conflict. In general, the snoop access has highest priority and is given first access to the tags. A pending load or store access will then occur on the clock following the snoop. However, the snoop is not given priority into the tags when the snoop coincides with a tag write (for example, validation after a cache block load). In these situations, the snoop is retried and must re-arbitrate before the lookup is possible.

Occasionally, cache snoops cannot be serviced and must be retried. These retries occur if the cache is busy with a burst read or write when the snoop operation takes place.

It is possible for a snoop to hit a modified cache block that is already in the process of being written to the copy-back buffer for replacement purposes. If this happens, the core retries the snoop, and raises the priority of the cast-out operation to allow it to occur on the CSB before the cache block fill.

4.10 Applications Information—Cache Locking

This section describes the entire cache locking and cache way-locking features of the e300 core.

4.10.1 Cache Locking Terminology

Cache locking refers to the ability to prevent some or all of a processor's instruction or data cache from being overwritten. Cache locking can be set for either an entire cache or for individual ways within the cache as follows:

- Entire cache locking—When an entire cache is locked, data for read hits within the cache are supplied to the requesting unit in the same manner as hits from an unlocked cache. Similarly, writes

that hit in the data cache are written to the cache in the same way as write hits to an unlocked cache. However, any access that misses in the cache is treated as a caching-inhibited access. Cache entries that are invalid at the time of locking remain invalid and inaccessible until the cache is unlocked. When the cache has been unlocked, all entries (including invalid entries) are available. Entire cache locking is inefficient if the number of instructions or the size of data to be locked is small compared to the cache size.

- Way-locking—Locking only a portion of the cache is accomplished by locking ways within the cache. Locking always begins with the first way (way 0) and is sequential; that is, locking ways 0, 1, and 2 is possible, but it is not possible to lock only way 0 and way 2. When using way-locking, at least one way must be left unlocked. The maximum number of lockable ways is seven on the e300c1 core (way 0–way 6) and three on the e300c2 and e300c3 (way 0–way 2).

  Unlike entire cache locking, invalid entries in a locked way are accessible and available for data replacement. As hits to the cache fill invalid entries within a locked way, the entries become valid and locked. This behavior differs from entire cache locking in which invalid entries cannot be allocated. Unlocked ways of the cache behave normally.

Table 4-10 summarizes the e300 core cache organization.

Table 4-10. Cache Organization

	Instruction Cache Size	Data Cache Size	Associativity	Block Size	Way Size
e300c1	32 Kbytes	32 Kbytes	8-way	8 words	4 Kbytes
e300c2	16 Kbytes	16 Kbytes	4-way	8 words	4 Kbytes
e300c3	16 Kbytes	16 Kbytes	4-way	8 words	4 Kbytes

4.10.2 Cache Locking Register Summary

Table 4-11 through Table 4-13 outline the registers and bits used to perform cache locking on the e300 core. Refer to Section 2.2.1, “Hardware Implementation Register 0 (HID0),” for a complete description of the HID0 and MSR registers. Refer to Section 2.2.3, “Hardware Implementation Register 2 (HID2),” for a complete description of the HID2 register.

Table 4-11. HID0 Bits Used to Perform Cache Locking

Bits	Name	Description
16	ICE	Instruction cache enable. This bit must be set for instruction cache locking. See Section 4.10.3.1.1, “Enabling the Data Cache.”
17	DCE	Data cache enable. This bit must be set for data cache locking. See Section 4.10.3.1.1, “Enabling the Data Cache.”
18	ILOCK	Instruction cache lock. Set to lock the entire instruction cache. See Section 4.10.3.2.5, “Locking the Entire Instruction Cache.”
19	DLOCK	Data cache lock. Set to lock the entire data cache. See Section 4.10.3.1.6, “Locking the Entire Data Cache.”

Table 4-11. HID0 Bits Used to Perform Cache Locking (continued)

Bits	Name	Description
20	ICFI	Instruction cache flash invalidate. Setting and then clearing this bit invalidates the entire instruction cache. See Section 4.10.3.2.7, "Invalidating the Instruction Cache (Even if Locked)."
21	DCFI	Data cache flash invalidate. Setting and then clearing this bit invalidates the entire data cache. See Section 4.10.3.1.4, "Invalidating the Data Cache."

Table 4-12. HID2 Bits Used to Perform Cache Way-locking

Bits	Name	Description
16–18	IWLCK	Instruction cache way-lock. These bits are used to lock individual ways in the instruction cache. See Section 4.10.3.2.6, "Way-Locking the Instruction Cache."
24–26	DWLCK	Data cache way-lock. These bits are used to lock individual ways in the data cache. See Section 4.10.3.1.7, "Way-Locking the Data Cache."

Table 4-13. MSR Bits Used to Perform Cache Locking

Bits	Name	Description
16	EE	External interrupt enable. This bit must be cleared during instruction and data cache loading. See Section 4.10.3.1.3, "Disabling Interrupts for Data Cache Locking."
19	ME	Machine check enable. This bit must be cleared during instruction and data cache loading. See Section 4.10.3.1.3, "Disabling Interrupts for Data Cache Locking."
26	IR	Instruction address translation. This bit must be set to enable instruction address translation by the MMU. See Section 4.10.3.1.2, "Address Translation for Data Cache Locking."
27	DR	Data address translation. This bit must be set to enable data address translation by the MMU. See Section 4.10.3.1.2, "Address Translation for Data Cache Locking."

4.10.3 Performing Cache Locking

This section outlines the basic procedures for locking the data and instruction caches and provides some example code for locking the caches. The procedures for the data cache are described first, followed by the corresponding sections for locking the instruction cache.

The basic procedures for cache locking are as follows:

- Enabling the cache
- Enabling address translation for example code
- Disabling interrupts
- Loading the cache
- Locking the cache (entire cache locking or cache way-locking)

In addition, this section describes how to invalidate the data and instruction caches, even when they are locked.

4.10.3.1 Data Cache Locking—Procedures

This section describes the procedures for performing data cache locking on the e300 core.

4.10.3.1.1 Enabling the Data Cache

To lock the data cache, the data cache enable bit HID0[DCE], bit 17, must be set. The following assembly code enables the data cache:

```
# Enable the data cache. This corresponds
# to setting DCE bit in HID0 (bit 17)

mfspr    r1, HID0
ori      r1, r1, 0x4000
sync
mtspr    HID0, r1
isync
```

4.10.3.1.2 Address Translation for Data Cache Locking

Two distinct memory areas must be set up to enable cache locking:

- The first area is where the code that performs the locking resides and is executed.
- The second area is where the data to be locked resides.

Both areas of memory must be in locations that are translated by the memory management unit (MMU). This translation can be performed either with the page table or the block address translation (BAT) registers.

For the purposes of the cache locking example in this document, the two areas of memory are defined using the BAT registers. The first area is a 1-Mbyte area in the upper region of memory that contains the code performing the cache locking. The second area is a 256-Mbyte block of memory (not all of the 256 Mbytes of memory is locked in the cache; this area is set up as an example) that contains the data to lock. Both memory areas use identity translation (the logical memory address equals the physical memory address).

Table 4-14 summarizes the BAT settings used in this example.

Table 4-14. Example BAT Settings for Cache Locking

Area	Base Address	Memory Size	WIMG Bits	BATU Setting	BATL Setting
First	0xFFF0_0000	1 Mbyte	0b0100[1]	0xFFF0_001F	0xFFF0_0002[1]
Second	0x0000_0000	256 Mbyte	0b0000	0x0000_1FFF	0x0000_0002

[1] Caching-inhibited memory is not a requirement for data cache locking. A setting of 0xFFF0_0002 with a corresponding WIMG of 0b0000 marks the memory area as caching-allowed.

The block address translation upper (BATU) and block address translation lower (BATL) settings in Table 4-14 can be used for both instruction block address translation (IBAT) and data block address translation (DBAT) registers. After the BAT registers have been set up, the MMU must be enabled. The following assembly code enables both instruction and data memory address translation:

```
# Enable instruction and data memory address translation. This
# corresponds to setting IR and DR in the MSR (bits 26 & 27)
```

```
        mfmsr    r1
        ori      r1, r1, 0x0030
        sync
        mtmsr    r1
        isync
```

4.10.3.1.3 Disabling Interrupts for Data Cache Locking

To ensure that interrupt handler routines do not execute while the cache is being loaded (which could possibly pollute the cache with undesired contents), all interrupts must be disabled. This is accomplished by clearing the appropriate bits in the machine state register (MSR). See Table 4-15 for the bits within the MSR that must be cleared to ensure that interrupts are disabled.

Table 4-15. MSR Bits for Disabling Interrupts

Bits	Name	Description
16	EE	External interrupt enable
19	ME	Machine check enable
20	FE0[1]	Floating-point exception mode 0
23	FE1[1]	Floating-point exception mode 1
24	CE	Critical interrupt enable

[1] The floating-point exception may not need to be disabled because the example code shown in this document that performs cache locking does not execute any floating-point operations.

The following assembly code disables all asynchronous interrupts:

```
# Clear the following bits from the MSR:
#    EE  (16)     ME  (19)
#    FE0 (20)     FE1 (23)
#    ME  (24)

        mfmsr    r1
        lis      r2, 0xFFFF
        ori      r2, r2, 0x667F
        and      r1, r1, r2
        sync
        mtmsr    r1
        isync
```

4.10.3.1.4 Invalidating the Data Cache

If a non-empty data cache has modified data, and the data cannot be discarded, the data cache must be flushed before it can be invalidated. Data cache flushing is accomplished by filling the data cache with known data and performing a flash invalidate or a series of **dcbf** instructions that force a flush and invalidation of the data cache block.

The following code sequence shows how to flush the data cache:

```
# r6 contains a block-aligned address in memory with which to fill
# the data cache. For this example, address 0x0 is used
        li      r6, 0x0

# CTR = number of data blocks to load
```

```
# e300c1 number of blocks = (32K) / (32 Bytes/block)
#                         = 2^15 / 2^5 = 2^9 = 0x400
# e300c2 and e300c3 number of blocks = (16K) / (32 Bytes/block)
#                         = 2^14 / 2^5 = 2^8 = 0x200

        li      r1, 0x400       # Use e300c1 value for this example
        mtctr   r1

# Save the total number of blocks in cache to r8
        mr      r8, r1

# Load the entire cache with known data
loop:   lwz     r2, 0(r6)
        addi    r6, r6, 32      # Find the next block
        bdnz    loop            # Decrement the counter, and
# branch if CTR != 0

# Now, flush the cache with dcbf instructions
        li      r6, 0x0         # Address of first block

        mtctr   r8              # Number of blocks
loop2:
        dcbf    r0, r6
        addi    r6, r6, 32      # Find the next block
        bdnz    loop2           # Decrement the counter, and
                                   #   branch if CTR != 0
```

If the content of the data cache does not need to be flushed to memory, the cache can be directly invalidated. The entire data cache is invalidated through the data cache flash invalidate bit HID0[DCFI], bit 21. Setting HID0[DCFI] and then immediately clearing it causes the entire data cache to be invalidated. The following assembly code invalidates the entire data cache (does not flush modified entries):

```
# Set and then clear the HID0[DCFI] bit, bit 21
        mfspr   r1, HID0
        mr      r2, r1
        ori     r1, r1, 0x0400
        sync
        isync
        mtspr   HID0, r1
        mtspr   HID0, r2
        isync
```

4.10.3.1.5 Loading the Data Cache

This section explains loading data into the data cache. The data cache can be loaded in several ways. The example in this document loads the data from memory. The following assembly code loads the data cache:

```
# Assuming interrupts are turned off, cache has been flushed,
# MMU on, and loading from contiguous caching-allowed memory.
# r6  = Starting address of code to lock
# r20 = Temporary register for loading into
# CTR = Number of cache blocks to lock

loop:   lwz     r20, 0(r6)      # Load data into d-cache
        addi    r6, r6, 32      # Find next block to load
        bdnz    loop            # CTR = CTR-1, branch if CTR != 0
```

4.10.3.1.6 Locking the Entire Data Cache

Locking of the entire data cache is controlled by the data cache lock bit (HID0[DLOCK], bit 19). Setting HID0[DLOCK] to 1 locks the entire data cache. To unlock the data, the HID0[DLOCK] must be cleared to 0. Setting the DLOCK bit must be preceded by a **sync** instruction to prevent the data cache from being locked during a data access. The following assembly code locks the entire data cache:

```
# Set the DLOCK bit in HID0 (bit 19)

        mfspr    r1, HID0
        ori      r1, r1, 0x1000
        sync
        mtspr    HID0, r1
        isync
```

NOTE

The **dcbz** instruction ignores DLOCK and always allocates a tag. Therefore, it is recommended that, when setting DLOCK the user also set the way-lock bits to lock the maximum number of ways. Refer to Section 4.10.3.1.7, "Way-Locking the Data Cache," for more information.

4.10.3.1.7 Way-Locking the Data Cache

Data cache way-locking is controlled by HID2[DWLCK], bits 24–26. Table 4-16 shows the HID2[DWLCK[0–2]] settings for the e300c1 core embedded processor.

Table 4-16. e300c1 Core DWLCK[0–2] Encodings

DWLCK[0:2]	Ways Locked
0b000	No ways locked
0b001	Way 0 locked
0b010	Ways 0 and 1 locked
0b011	Ways 0, 1, and 2 locked
0b100	Ways 0, 1, 2, and 3 locked
0b101	Ways 0, 1, 2, 3, and 4 locked
0b110	Ways 0, 1, 2, 3, 4, and 5 locked
0b111	Ways 0, 1, 2, 3, 4, 5, and 6 locked

Table 4-16 shows the HID2[DWLCK[0–2]] settings for the e300c2 and e300c3 core embedded processor.

Table 4-17. e300c2 and e300c3 Core DWLCK[0–2] Encodings

DWLCK[0:2]	Ways Locked
0b000	No ways locked
0b001	Way 0 locked
0b010	Ways 0 and 1 locked
0b011	Ways 0, 1, and 2 locked
0b100	Reserved (Ways 0, 1, and 2 locked)
0b101	Reserved (Ways 0, 1, and 2 locked)
0b110	Reserved (Ways 0, 1, and 2 locked)
0b111	Reserved (Ways 0, 1, and 2 locked)

Note that on the e300c2 and e300c3, values greater than 0b011 are reserved but default to the maximum number of ways locked (Ways 0,1, and 2).

The following assembly code locks way 0 of the e300 core data cache:

```
# Lock way 0 of the data cache
# This corresponds to setting dwlck(0-2) 0b001 (bits 24-26)

        mfspr    r1, HID2
        lis      r2, 0xFFFF
        ori      r2, r2, 0xFF1F
        and      r1, r1, r2
        ori      r1, r1, 0x0020
        sync
        mtspr    HID2, r1
        isync
```

4.10.3.1.8 Invalidating the Data Cache (Even if Locked)

There are two methods for invalidating the instruction or data cache:

- Invalidate the entire cache by setting and then immediately clearing the data cache flash invalidate bit HID0[DCFI], bit 21. Even when a cache is locked, toggling the DCFI bit invalidates all of the data cache.
- The data cache block invalidate (**dcbi**) instruction can be used to invalidate individual cache blocks.

4.10.3.2 Instruction Cache Locking—Procedures

This section describes the procedures for performing instruction cache locking on the e300 core.

4.10.3.2.1 Enabling the Instruction Cache

To lock the instruction cache, the instruction cache enable bit HID0[ICE], bit 16 must be set.

```
# Enable the data cache. This corresponds
# to setting DCE bit in HID0 (bit 17)

        mfspr    r1, HID0
        ori      r1, r1, 0x8000
        sync
        mtspr    HID0, r1
        isync
```

4.10.3.2.2 Address Translation for Instruction Cache Locking

Two distinct memory areas must be set up to enable cache locking:

- The first area is where the code that performs the locking resides and is executed.
- The second area is where the instructions to be locked reside.

Both areas of memory must be in locations that are translated by the memory management unit (MMU). This translation can be performed either with the page table or the block address translation (BAT) registers.

For the purposes of the cache locking example in this document, two areas of memory are defined using the BAT registers. The first area is a 1-Mbyte area in the upper region of memory that contains the code performing the cache locking. This area of memory must be caching-inhibited for instruction cache locking. The second area is a 256-Mbyte block of memory that contains the instructions to lock (not all of the 256 Mbytes of memory is locked in the cache; this area is set up as an example). Both memory areas use identity translation (the logical memory address equals the physical memory address). Table 4-18 summarizes the BAT settings used in this example.

Table 4-18. Example BAT Settings for Cache Locking

Area	Base Address	Memory Size	WIMG Bits	BATU Setting	BATL Setting
First	0xFFF0_0000	1 Mbyte	0b0100 [1]	0xFFF0_001F	0xFFF0_0022[1]
Second	0x0000_0000	256 Mbytes	0b0000	0x0000_1FFF	0x0000_0002

[1] 0xFFF0_0022 defines a caching-inhibited memory area used for instruction cache locking and corresponds to a WIMG of 0b0100. Caching-inhibited memory is not a requirement for data cache locking. A setting of 0xFFF0_0002 with a corresponding WIMG of 0b0000 marks the memory area as caching-allowed.

The block address translation upper (BATU) and block address translation lower (BATL) settings in Table 4-18 can be used for both instruction block address translation (IBAT) and data block address translation (DBAT) registers. After the BAT registers have been set up, the MMU must be enabled.

The following assembly code enables both instruction and data memory address translation:

```
# Enable instruction and data memory address translation. This
# corresponds to setting IR and DR in the MSR (bits 26 & 27)

        mfmsr    r1
        ori      r1, r1, 0x0030
```

```
        sync
        mtmsr    r1
        isync
```

4.10.3.2.3 Disabling Interrupts for Instruction Cache Locking

To ensure that interrupt handler routines do not execute while the cache is being loaded (which could possibly pollute the cache with undesired contents) all interrupts must be disabled. This is accomplished by clearing the appropriate bits in the machine state register (MSR). See Table 4-19 for the bits within the MSR that must be cleared to ensure that interrupts are disabled.

Table 4-19. MSR Bits for Disabling Interrupts

Bit	Name	Description
16	EE	External interrupt enable
19	ME	Machine check enable
20	FE0[1]	Floating-point exception mode 0
23	FE1 [1]	Floating-point exception mode 1
24	CE	Critical interrupt enable

[1] The floating-point exception may not need to be disabled because the example code shown in this document that performs cache locking does not execute any floating-point operations.

The following assembly code disables all asynchronous interrupts:

```
# Clear the following bits from the MSR:
#    EE  (16)     ME  (19)
#    FE0 (20)     FE1 (23)
#        ME (24)

        mfmsr    r1
        lis      r2, 0xFFFF
        ori      r2, r2, 0x667F
        and      r1, r1, r2
        sync
        mtmsr    r1
        isync
```

4.10.3.2.4 Preloading Instructions into the Instruction Cache

To optimize performance, processors that implement the PowerPC architecture automatically prefetch instructions into the instruction cache. This feature can be used to preload explicit instructions into the cache even when it is known that their execution will be canceled. Although the execution of the instructions is canceled, the instructions remain valid in the instruction cache.

Because instructions are intentionally executed speculatively, care must be taken to ensure that all I/O memory is marked guarded. Otherwise, speculative loads and stores to I/O space could potentially cause data loss. See the *Programming Environments Manual* for a full discussion of guarded memory.

The code that prefetches must be in caching-inhibited memory as in the following example:

```
# Assuming interrupts are disabled, cache has been flushed,
# the MMU is on, and we are executing in a caching-inhibited
```

```
# location in memory
# LR and r6 = Starting address of code to lock
# CTR = Number of cache blocks to lock
# r2 = non-zero numerator and denominator
# ‘loop’ must begin on an 8-byte boundary to ensure that
#    the divw and beqlr+ are fetched on the same cycle.

.orig    0xFFF04000

loop:    divw.    r2, r2, r2        # LONG divide w/ non-zero result
         beqlr+                     # Cause the prefetch to happen

         addi     r6, r6, 32        # Find next block to prefetch
         mtlr     r6                # set the next block
         bdnz-    loop              # Decrement the counter and
                                    # branch if CTR != 0
```

In the above example, both the **divw** and **beqlr+** instructions are fetched at the same time (this assumes a 64-bit data bus; the preloading code does not work for a 32-bit data bus) due to their placement on a double-word boundary. The divide instruction was chosen because it takes many cycles to execute. During execution of the divide, the processor starts fetching instructions speculatively at the target destination of the branch instruction. The speculation occurs because the branch is statically predicted as taken. This speculative fetching causes the cache block that is pointed to by the link register (LR) to be loaded into the cache. Because the **divw.** instruction always produces a non-zero result, the **beqlr+** is not taken and execution of all speculatively fetched instructions is canceled. However, the instructions remain valid in the cache.

If the destination instruction stream contains an unconditional branch to another memory location, it is possible to also prefetch the destination of the unconditional branch instruction. This does not cause a problem if the destination of the unconditional branch is also inside the area of memory that needs to be preloaded. But if the destination of the unconditional branch is not in the area of memory to be loaded, then care must be taken to ensure that the branch destination is to an area of memory that is caching-inhibited. Otherwise, unintentional instructions may be locked in the cache and the desired instructions may not be in their expected way within the cache.

The **icbt** is a new feature to the e300 core and supplements the instruction cache locking mechanisms and the new way-protect feature. The **icbt** instruction performs a bus read operation from the bus and allocates into the instruction cache.

The **icbt** instruction functions independently of WIMG settings, instruction or data cache enable status, or the instruction cache lock status (i.e. unconditional allocate). This allows the **icbt** instruction to be used to easily initialize the locked portion of the instruction cache before enabling.

Note that to prevent user-mode code from inadvertently over-writing a supervisor-mode page that has been locked in the instruction cache, that page of memory should be protected with appropriate MMU translation and access privileges, or HID0[NOPTI] should be set in order to no-op further uses of the instruction.

The **icbt** instruction is dispatched to the load/store unit and data cache, and therefore goes through data-side address translation. It is treated similarly to **dcbt** for translation and storage protection purposes. The data cache then issues the **icbt** operation to the bus unit without checking for a data cache hit.

Therefore if the instruction block may already be residing in the data cache (because of self-modifying code, for example), the program must first perform a **dcbf** or **dcbst** of that address to flush that data block back to main memory for the **icbt** to access it.

A **sync** instruction must follow an **icbt** instruction to ensure all cache load operations are completed. An **isync** instruction must also follow an **icbt** if instruction fetching to the newly touched and loaded instructions may occur immediately after the **icbt** was executed.

4.10.3.2.5 Locking the Entire Instruction Cache

Locking the entire instruction cache is controlled by the instruction cache lock bit (HID0[ILOCK], bit 18). Setting HID0[ILOCK] locks the entire instruction cache, and clearing HID0[ILOCK] allows the instruction cache to operate normally. The setting of the HID0[ILOCK] should be preceded by an **isync** instruction to prevent the instruction cache from being locked during an instruction access. The following assembly code locks the contents of the entire instruction cache.

```
# Set the ILOCK bit in HID0 (bit 18)

        mfspr    r1, HID0
        ori      r1, r1, 0x2000
        sync
        isync
        mtspr    HID0, r1
        isync
```

4.10.3.2.6 Way-Locking the Instruction Cache

Instruction cache way-locking is controlled by the HID2[IWLCK], bits 16–18. Table 4-20 shows the HID2[IWLCK[0–2]] settings for the core.

Table 4-20. e300c1 **Core IWLCK[0–2] Encodings**

IWLCK[0:2]	Ways Locked
0b000	No ways locked
0b001	Way 0 locked
0b010	Ways 0 and 1 locked
0b011	Ways 0, 1, and 2 locked
0b100	Ways 0, 1, 2, and 3 locked
0b101	Ways 0, 1, 2, 3, and 4 locked
0b110	Ways 0, 1, 2, 3, 4, and 5 locked
0b111	Ways 0, 1, 2, 3, 4, 5, and 6 locked

Table 4-16 shows the HID2[IWLCK[0–2]] settings for the e300c2 and e300c3.

Table 4-21. e300c2 Core IWLCK[0–2] Encodings

IWLCK[0:2]	Ways Locked
0b000	No ways locked
0b001	Way 0 locked
0b010	Ways 0 and 1 locked
0b011	Ways 0, 1, and 2 locked
0b100	Reserved (Ways 0, 1, and 2 locked)
0b101	Reserved (Ways 0, 1, and 2 locked)
0b110	Reserved (Ways 0, 1, and 2 locked)
0b111	Reserved (Ways 0, 1, and 2 locked)

Note that on the e300c2 and e300c3, values greater than 0b011 are reserved but default to the maximum number of ways locked (Ways 0,1, and 2).

The following assembly code locks way 0 of the core instruction cache:

```
# Lock way 0 of the instruction cache
# This corresponds to setting iwlck(0–2) to 0b001 (bits 16–18)

        mfspr    r1, HID2
        lis      r2, 0xFFFF
        ori      r2, r2, 0x1FFF
        and      r1, r1, r2
        ori      r1, r1, 0x2000
        sync
        isync
        mtspr    HID2, r1
        isync
```

4.10.3.2.7 Invalidating the Instruction Cache (Even if Locked)

There are two methods for invalidating the instruction cache. In the first way, invalidate the entire cache by setting and then immediately clearing the instruction cache flash invalidate bit (HID0[ICFI], bit 20). Even when a cache is locked, toggling the ICFI bit invalidates all of the instruction cache. The following assembly code invalidates the entire instruction cache:

```
# Set and then clear the HIDO[ICFI] bit, bit 20

        mfspr    r1, HID0
        mr       r2, r1
        ori      r1, r1, 0x0800
        sync
        isync
        mtspr    HID0, r1
        mtspr    HID0, r2
        isync
```

In the second method, the instruction cache block invalidate (**icbi)** instruction can be used to invalidate individual cache blocks. The **icbi** instruction invalidates blocks in an entirely locked instruction cache. The **icbi** instruction also may invalidate way-locked blocks within the instruction cache.

4.10.3.2.8 Instruction Cache Way Protection

The instruction cache way-protect extension supplements the existing cache lock capability of the instruction cache. For normal operation, from one to all ways (eight ways for 32Kbyte caches/four ways for 16Kbyte caches) of the instruction cache may be locked using HID2[IWLCK] and HID0[ILOCK]. The locking mechanism normally allows a portion or all of the instruction cache to be used as a very fast, always resident, program memory. Specifically, pages of memory (up to a full 4Kbytes per page per way) may be locked in the instruction cache and used as fast, always-resident program memory, while allowing the remaining unlocked portion of the instruction cache to continue to operate as normal cache for the remaining program.

Normal instruction cache management, however, typically relies on the **icbi** instruction and the flash invalidate mechanism (HID0[ICFI]) to routinely clear out part or all of the instruction cache for program-related operations, such as process changes and MMU page deallocation. These cache management operations clear out both the locked and unlocked portions of the cache (**icbi** invalidates all ways of a cache set unconditionally, and HID0[ICFI] clears the entire instruction cache unconditionally).

In the e300 core, the new instruction cache way protect extension prevents the locked portion of the instruction cache from being invalidated by the **icbi** instruction or by the flash invalidate mechanism (HID0[ICFI]). This allows the locked portion of the instruction cache to have the same persistence as main memory, while still allowing the remaining unlocked portion of the instruction cache to be managed by the program. This way-protect extension is enabled by setting HID2[ICWP].

In addition to the way-protect extension, the instruction cache block touch (**icbt**) instruction is added to support easy start-up initialization or reloading of the instruction cache. This specifically supports the way-lock and way-protect mechanism by allowing the locked portion of the cache to be easily initialized in a predictable fashion. (The natural prefetch mechanism of super-scalar processors otherwise precludes this.)

The **icbt** instruction allocates blocks into the instruction cache, regardless of WIMG settings or whether the instruction cache is enabled, so that the instruction cache can be easily locked and preloaded before turning on. In addition, in case of a cache hit, the **icbt** instruction will re-write to the same hitting line, so that an existing locked and protected page in the instruction cache can be easily re-loaded with a different program without going back to an initial start-up state.

Chapter 5
Interrupts and Exceptions

The PowerPC interrupt mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When interrupts occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (interrupt vector) predetermined for each interrupt. Processing of interrupts occurs in supervisor mode.

Although multiple exception conditions can map to a single interrupt vector, a more specific condition may be determined by examining a register associated with the interrupt or exception that causes it—for example, the DSISR or FPSCR. Additionally, certain exception conditions and interrupts can be explicitly enabled or disabled by software.

The PowerPC architecture requires that interrupts be handled in program order; therefore, although a particular implementation may recognize interrupt conditions out of order, they are handled strictly in order with respect to the instruction stream. When an instruction-caused interrupt is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the interrupt is taken. Any interrupts caused by those instructions are handled first. Likewise, interrupts that are asynchronous and precise are recognized when they occur, but are not handled until the instruction currently in the completion stage successfully completes execution or generates an interrupt, and the completed store queue is emptied (see Section 7.1, "Terminology and Conventions," for the definition). An instruction is said to have completed when the results of that instruction's execution have been committed to the registers defined by the architecture (for example, the GPRs or FPRs, rather than rename buffers). If a single instruction encounters multiple exception conditions, those exceptions are taken and handled sequentially. Likewise, interrupts that are asynchronous are recognized when they occur, but are not handled until the next instruction to complete in program order successfully completes. Throughout this chapter, the phrase 'next instruction' implies the next instruction to complete in program order.

Note that interrupts can occur while an interrupt handler routine is executing, and multiple interrupts can become nested. It is up to the interrupt handler to save the states to allow control to ultimately return to the original excepting program.

Unless a catastrophic condition causes a system reset or machine check interrupt, only one interrupt is handled at a time. If, for example, a single instruction encounters multiple interrupt conditions, those conditions are handled sequentially. After the interrupt handler is finished, instruction execution continues until the next interrupt is encountered. However, in many cases there is no attempt to re-execute the instruction. This method of recognizing and handling interrupts sequentially guarantees that interrupts are recoverable.

To prevent loss of state information, interrupt handlers should save the information stored in SRR0 and SRR1 (or in CSRR0/CSRR1 for critical interrupts) soon after the interrupt is taken. This prevents loss of

information due to a system reset or machine check interrupt or to an instruction-caused interrupt in the interrupt handler before disabling external interrupts.

In this chapter, the following terminology is used to describe the various stages of exception processing:

Exception	A condition that, if enabled, generates an interrupt.
Recognition	Interrupt recognition occurs when the exception that can cause an interrupt is identified by the processor.
Taken	An interrupt is said to be taken when control of instruction execution is passed to the interrupt handler; that is, the context is saved and the instruction at the appropriate vector offset is fetched and the interrupt handler routing is executed in supervisor mode.
Handling	Interrupt handling is performed by the software linked to the appropriate vector offset. Interrupt handling is performed at the supervisor-level.

5.1 Interrupt Classes

The PowerPC architecture supports four types of interrupts:

- Synchronous, precise—These are caused by instructions. All instruction-caused interrupts are handled precisely; that is, the machine state at the time the interrupt occurs is known and can be completely restored. This means that (excluding the trap and system call interrupts) the address of the faulting instruction is provided to the interrupt handler and that neither the faulting instruction nor subsequent instructions in the code stream will complete execution before the interrupt is taken. Once the interrupt is processed, execution resumes at the address of the faulting instruction (or at an alternate address provided by the interrupt handler). When an interrupt is taken due to a trap or system call instruction, execution resumes at an address provided by the handler.
- Synchronous, imprecise—The PowerPC architecture defines two imprecise floating-point exception modes: recoverable and nonrecoverable. Even though the e300 core provides a means to enable the imprecise modes, it implements these modes identically to the precise mode (that is, all enabled floating-point exceptions are always precise on the e300 core). These are not implemented on the e300c2 core as it does not support floating-point instructions.
- Asynchronous, maskable—The external interrupt ($\overline{int}$), system management interrupt ($\overline{smi}$), decrementer interrupt, and critical interrupt ($\overline{cint}$) are maskable asynchronous interrupts. When these interrupts occur, their handling is postponed until the next instruction completes execution and until any interrupts associated with that instruction complete execution. If there are no instructions in the execution units, the interrupt is taken immediately upon determination of the correct restart address (for loading SRR0).
- Asynchronous, nonmaskable—There are two nonmaskable asynchronous interrupts: system reset and the machine check interrupt. These interrupts may not be recoverable, or may provide a limited degree of recoverability. All interrupts report recoverability through the MSR[RI] bit.

The e300 core interrupt classes are shown in Table 5-1.

Table 5-1. Interrupt Classifications

Synchronous/Asynchronous	Precise/Imprecise	Interrupt Type
Asynchronous, nonmaskable	Imprecise	Machine check System reset
Asynchronous, maskable	Precise	External interrupt Decrementer System management interrupt Critical interrupt
Synchronous	Precise	Instruction-caused interrupts

Table 5-1 defines interrupt categories that are handled uniquely by the e300 core. Note that Table 5-1 includes no synchronous imprecise interrupts. While the PowerPC architecture supports imprecise handling of floating-point exceptions, the e300 core implements them as precise. Although the PowerPC architecture specifies that the recognition of the machine check interrupt is nonmaskable, on the e300 core the stimuli that cause this interrupt are maskable. For example, the machine check interrupt is caused by the assertion of $\overline{tea}$, $\overline{ape}$, $\overline{dpe}$, or $\overline{mcp}$. However, $\overline{mcp}$, $\overline{ape}$, and $\overline{dpe}$ can be disabled by bits 0, 2, and 3, respectively, in HID0. Therefore, the machine check caused by asserting $\overline{tea}$ is the only truly nonmaskable machine check interrupt.

The e300 core interrupts, and conditions that cause them, are listed in Table 5-2.Note that the e300c2 core does not support floating-point operations.

Table 5-2. Interrupts and Exception Conditions

Interrupt Type	Vector Offset (hex)	Exception Conditions
Reserved	00000	—
System reset	00100	A system reset is caused by the assertion of either $\overline{sreset}$ or $\overline{hreset}$.
Machine check	00200	A machine check is caused by the assertion of the $\overline{tea}$ signal during a data bus transaction, assertion of $\overline{mcp}$, an address or data parity error, or an instruction or data cache parity error. Note that when a machine check occurs, the e300 has SRR1 register values that are different from the G2/G2_LE cores because the e300 core supports cache parity. See Table 5-14 for more information.
DSI	00300	The cause of a DSI interrupt can be determined by the bit settings in the DSISR, listed as follows: 1 Set if the translation of an attempted access is not found in the primary hash table entry group (HTEG), or in the rehashed secondary HTEG, or in the range of a DBAT register; otherwise cleared. 4 Set if a memory access is not permitted by the page or DBAT protection mechanism; otherwise cleared. 6 Set for a store operation and cleared for a load operation 9 Set if a data address breakpoint interrupt occurs when the data [0–28] in the DABR or DABR2 matches the next data access (load or store instruction) to complete in the completion unit. The different breakpoints are enabled as follows: • Write breakpoints enabled when DABR[30] is set • Read breakpoints enabled when DABR[31] is set

Table 5-2. Interrupts and Exception Conditions (continued)

Interrupt Type	Vector Offset (hex)	Exception Conditions
ISI	00400	An ISI interrupt is caused when an instruction fetch cannot be performed for any of the following reasons: • The effective (logical) address cannot be translated. That is, there is a page fault for this portion of the translation, so an ISI interrupt must be taken to load the PTE (and possibly the page) into memory. • The fetch access violates memory protection (indicated by SRR1[4] set). If the key bits (Ks and Kp) in the segment register and the PP bits in the PTE are set to prohibit read access, instructions cannot be fetched from this location.
External interrupt	00500	An external interrupt is caused when MSR[EE] = 1 and the $\overline{int}$ signal is asserted.
Alignment	00600	An alignment interrupt is caused when the core cannot perform a memory access for any of the reasons described below: • The operand of a floating-point load or store instruction is not word-aligned. • The operands of **lmw**, **stmw**, **lwarx**, and **stwcx.** instructions are not aligned. • The operand of a load, store, load multiple, store multiple, load string, or store string instruction crosses a protection boundary. • The instruction is **lswi**, **lswx**, **stswi**, **stswx**, and the core is in little-endian mode. Note that PowerPC little-endian mode is not supported on the e300 core. • The operand of **dcbz** is in memory that is write-through-required or caching-inhibited.
Program	00700	A program interrupt is caused by one of the following exceptions, which correspond to bit settings in SRR1 and arise during execution of an instruction. Floating-point enabled exception—A floating-point enabled exception condition is generated when the following condition is met: (MSR[FE0] | MSR[FE1]) & FPSCR[FEX] is 1. • FPSCR[FEX] is set by the execution of a floating-point instruction that causes an enabled exception or by the execution of one of the ‘move to FPSCR’ instructions that results in both an exception condition bit and its corresponding enable bit being set in the FPSCR. • Illegal instruction—An illegal instruction program exception is generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields (including PowerPC instructions not implemented in the core), or when execution of an optional instruction not provided in the core is attempted (these do not include those optional instructions that are treated as no-ops). • Privileged instruction—A privileged instruction type program exception is generated when the execution of a privileged instruction is attempted and the MSR register user privilege bit, MSR[PR], is set. In the e300 core, this exception is generated for **mtspr** or **mfspr** with an invalid SPR field if SPR[0] = 1 and MSR[PR] = 1. This may not be true for all cores that implement the PowerPC architecture. • Trap—A trap type program exception is generated when any of the conditions specified in a trap instruction is met.
Floating-point unavailable	00800	A floating-point unavailable interrupt is caused by an attempt to execute a floating-point instruction (including floating-point load, store, and move instructions) when the floating-point available bit is cleared (MSR[FP] = 0).
Decrementer	00900	The decrementer interrupt occurs when DEC[0] changes from 0 to 1. This interrupt is enabled with MSR[EE].
Critical interrupt	00A00	A critical interrupt is taken when $\overline{cint}$ is asserted and MSR[CE] = 1.
Reserved	00B00–00BFF	—
System call	00C00	A system call interrupt occurs when a System Call (**sc**) instruction is executed.

Table 5-2. Interrupts and Exception Conditions (continued)

Interrupt Type	Vector Offset (hex)	Exception Conditions
Trace	00D00	A trace interrupt is taken when MSR[SE] =1 or when the currently completing instruction is a branch and MSR[BE] =1.
Reserved	00E00	The e300 core does not generate an interrupt to this vector. Other devices may use this vector for floating-point assist interrupts.
Performance monitor	00F00	Caused when *pm_event_in* is asserted.
Instruction translation miss	01000	An instruction translation miss interrupt is caused when the effective address for an instruction fetch cannot be translated by the ITLB.
Data load translation miss	01100	A data load translation miss interrupt is caused when the effective address for a data load operation cannot be translated by the DTLB.
Data store translation miss	01200	A data store translation miss interrupt is caused when the effective address for a data store operation cannot be translated by the DTLB, or where a DTLB hit occurs, and the change bit in the PTE must be set due to a data store operation.
Instruction address breakpoint	01300	An instruction address breakpoint interrupt occurs when the address (bits 0–29) in the IABR matches the next instruction to complete in the completion unit, and IABR[30] is set. Note that the e300 core also implements IABR2, which functions identically to IABR.
System management interrupt	01400	A system management interrupt is caused when MSR[EE] = 1 and the $\overline{smi}$ input signal is asserted.
Reserved	01500–02FFF	—

Interrupts are roughly prioritized by class, as follows:

1. Nonmaskable, asynchronous interrupts have priority over all other interrupts— system reset and machine check interrupts (although the machine check exception conditions can be disabled so the condition causes the processor to go directly into the checkstop state). These interrupts cannot be delayed and do not wait for the completion of any precise interrupt handling.
2. Synchronous, precise interrupts are caused by instructions and are taken in strict program order.
3. Maskable asynchronous interrupts (for example, external interrupt and decrementer interrupts) are delayed until higher priority interrupts are taken.

System reset and machine check interrupts may occur at any time and are not delayed even if an interrupt is being handled. As a result, state information for the interrupted interrupt may be lost; therefore, these interrupts are typically nonrecoverable.

All other interrupts have lower priority than system reset and machine check interrupts, and the interrupt may not be taken immediately when it is recognized.

5.1.1 Interrupt Priorities

The interrupts are listed in Table 5-3 in order of highest to lowest priority.

Table 5-3. Interrupt Priorities

Interrupt Category	Priority	Interrupt	Cause
Asynchronous	0	System reset	*hreset*
	1	Machine check	*tea*, *mcp*, *ape*, or *dpe*
	2	System reset	*sreset*
	3	Critical interrupt	*cint*. See Section 5.2.1.2, "CSRR0 and CSRR1 Bit Settings," for more information
	4	System management interrupt	*smi*
	5	External interrupt	*int*
	6	Performance monitor interrupt	*pm_event_in*
	7	Decrementer interrupt	Decrementer passed through 0x0000_0000
Instruction fetch	0	ITLB miss	Instruction TLB miss
	1	Instruction access	Instruction access interrupt
Instruction dispatch/ execution	0	IABR	Instruction address breakpoint interrupt
	1	Program	Program interrupt due to the following: • Illegal instruction • Privileged instruction • Trap
	2	System call	System call interrupt
	3	Floating-point unavailable	Floating-point unavailable interrupt
	4	Program	Floating-point enabled interrupt
	5	Alignment	One to the following: • Floating-point not word-aligned • **lmw**, **stmw**, **lwarx**, or **stwcx.** not word-aligned • **ecwix** or **ecowx** operands not aligned • String access with little-endian bit (MSR[LE]) set • The operand of a load, store, load multiple, store multiple, load string, or store string instruction crosses a protection boundary. • The instruction is **lswi**, **lswx**, **stswi**, **stswx**, and the core is in little-endian mode. Note that PowerPC little-endian mode is not supported on the e300 core.
	6	Data access	BAT page protection violation
	7	DTLB miss	A store or load miss
	8	Alignment	A **dcbz** to a write-through or caching-inhibited page
	9	Data access	A TLB page protection violation
	10	DTLB miss	A change bit not set on a store operation
Post-instruction execution	0	Trace	One of the following: • MSR[SE] = 1 • MSR[BE] = 1 for branches

Interrupt priorities are described in detail in “Interrupt Priorities,” in Chapter 6, “Interrupts,” in the *Programming Environments Manual*.

5.1.2 Summary of Front-End Interrupt Handling

The following list of interrupt categories describes how the e300 core handles interrupts up to the point of signaling the appropriate interrupt to occur. Note that a recoverable state is reached if the completed store queue is empty (drained, not canceled) and any instruction that is next in program order and has been signaled to complete has completed. If MSR[RI] is clear, the core is in a nonrecoverable state by default. Also, completion of an instruction is defined as performing all architectural register writes associated with that instruction, and then removing that instruction from the completion buffer queue.

- Asynchronous nonmaskable nonrecoverable—(system reset caused by the assertion of $\overline{\textit{hreset}}$). These interrupts have highest priority and are taken immediately regardless of other pending interrupts or recoverability. A nonpredicted address is guaranteed.
- Asynchronous maskable nonrecoverable—(machine check). A machine check interrupt takes priority over any other pending interrupt except a nonrecoverable system reset caused by the assertion of either $\overline{\textit{hreset}}$ or internally during POR. A machine check interrupt is taken immediately regardless of recoverability. A machine check interrupt can occur only if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs. A nonpredicted address is guaranteed.
- Asynchronous nonmaskable recoverable—(system reset caused by the assertion of $\overline{\textit{sreset}}$). This interrupt takes priority over any other pending interrupts except nonrecoverable interrupts listed above. This interrupt is taken immediately when a recoverable state is reached.
- Asynchronous maskable recoverable—(system management interrupt, critical interrupt, external interrupt, decrementer interrupt). Before handling this type of interrupt, the next instruction in program order must complete or except. If this action causes another type of interrupt, that interrupt is taken and the asynchronous maskable recoverable interrupt remains pending. Once an instruction can complete without causing an interrupt, further instruction completion is halted while the interrupt not taken remains pending. The interrupt is taken when a recoverable state is reached.
- Instruction fetch—(ITLB, ISI). When this type of interrupt is detected, dispatch is halted and the current instruction stream is allowed to drain. If completing any instructions in this stream causes an interrupt, that interrupt is taken and the instruction fetch interrupt is forgotten. Otherwise, as soon as the machine is empty and a recoverable state is reached, the instruction fetch interrupt is taken.
- Instruction dispatch/execution—(program, DSI, alignment, emulation trap, system call, DTLB miss on load or store, IABR). This type of interrupt is determined at dispatch or execution of an instruction. The interrupt remains pending until all instructions in program order before the interrupt-causing instruction are completed. The interrupt is then taken without completing the interrupt-causing instruction. If any other exception condition is created in completing these previous instructions in the machine, that interrupt takes priority over the pending instruction dispatch/execution interrupt, which is then forgotten.
- Post-instruction execution—(trace). This type of interrupt is generated following execution and completion of an instruction while a trace mode is enabled. If executing the instruction produces

conditions for another type of interrupt, that interrupt is taken and the post-instruction execution interrupt is forgotten for that instruction.

5.2 Interrupt Processing

When an interrupt is taken, the processor uses the save/restore registers, SRR0 and SRR1, to save the contents of the machine state register for user-level mode and to identify where instruction execution should resume after the interrupt is handled.

5.2.1 Interrupt Processing Registers

The e300 core implements the SRR0 and SRR1 registers that are used for saving processor state on an interrupt. Additionally, the e300 core implements CSRR0 and CSRR1 to specifically save state for critical interrupt interrupts.

5.2.1.1 SRR0 and SRR1 Bit Settings

When an interrupt occurs, SRR0 is set to point to the instruction at which instruction processing should resume when the interrupt handler returns control to the interrupted process. All instructions in the program flow preceding this one will have completed and no subsequent instruction will have completed. This may be the address of the instruction that caused the interrupt or the next one (as in the case of a system call interrupt). The instruction addressed can be determined from the interrupt type and status bits. This address is used to resume instruction processing in the interrupted process, typically when an **rfi** instruction is executed. The SRR0 register is shown in Figure 5-1.

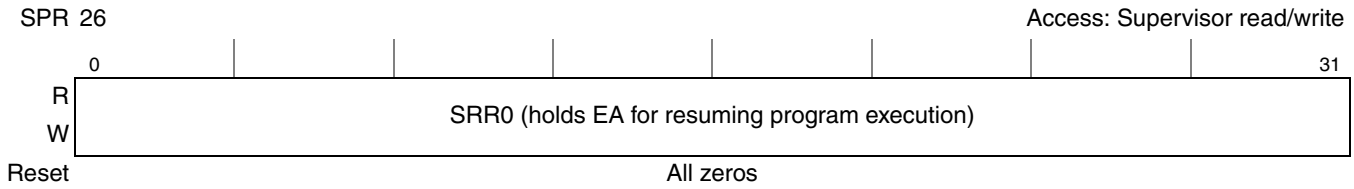


Figure 5-1. Machine Status Save/Restore Register 0 (SRR0)

The save/restore register 1 (SRR1) is used to save machine status (the contents of the MSR) on interrupts and to restore those values when **rfi** is executed. SRR1 is shown in Figure 5-2.

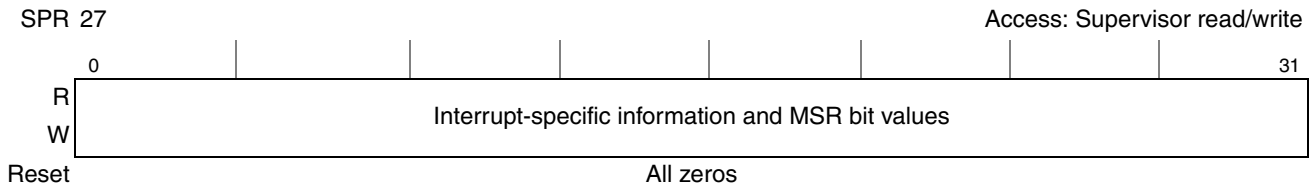


Figure 5-2. Machine Status Save/Restore Register 1 (SRR1)

Typically, when an interrupt occurs, SRR1[0–15] are loaded with interrupt-specific information and bits 16–31 of MSR are placed into the corresponding bit positions of SRR1. The e300 core loads SRR1 with specific bits for handling machine check interrupts, as shown in Table 5-4.

Table 5-4. SRR1 Bit Settings for Machine Check Interrupts

Bits	Name	Description
0–9	—	Cleared
10	ICPE	Instruction cache parity error
11	DCPE	Data cache parity error
12	MCP	Machine check
13	TEA	TEA error
14	DPE	Data parity error
15	APE	Address parity error
16–29	MSR[16–29]	Copy of MSR bits16–29
30	MSR[30]	Cleared for instruction cache parity error, data cache parity error, $\overline{tea}$, $\overline{dpe}$, $\overline{ape}$; copied from MSR[30] for $\overline{mcp}$. If $\overline{mcp}$ and $\overline{tea}$ are asserted simultaneously, then SRR1[30] is cleared and the interrupt is not recoverable.
31	MSR[31]	Copy of MSR[31]

The e300 core loads SRR1 with specific bits for handling program interrupts, as shown in Table 5-5.

Table 5-5. SRR1 Bit Settings for Program Interrupts

Bits	Name	Description
0–10	—	Cleared
11	—	Floating-point enabled program exception. Otherwise cleared.
12	—	Illegal instruction program exception. Otherwise cleared.
13	—	Privileged instruction program exception. Otherwise cleared.
14	—	Trap program exception. Otherwise cleared.
15	—	Set if SRR0 contains the address of a subsequent instruction. Cleared if SRR0 contains the address of the instruction causing the exception condition.
16–29	MSR[16–29]	Copy of MSR bits16–29
30	MSR[30]	Cleared for instruction cache parity error, data cache parity error, $\overline{tea}$, $\overline{dpe}$, $\overline{ape}$; copied from MSR[30] for $\overline{mcp}$. If $\overline{mcp}$ and $\overline{tea}$ are asserted simultaneously, then SRR1[30] is cleared and the interrupt is not recoverable.
31	MSR[31]	Copy of MSR[31]

The e300 core loads SRR1 with specific bits for handling the three TLB miss interrupts, as shown in Table 5-6.

Table 5-6. SRR1 Bit Settings for Software Table Search Operations

Bits	Name	Description
0–3	CRF0	Copy of condition register field 0 (CR0)
4	—	Reserved
5–9	MSR[5–9]	Copy of MSR bits 5–9
10–11	—	Reserved
12	KEY	TLB miss protection key
13	I/D	Instruction/data TLB miss 0 DTLB miss 1 ITLB miss
14	WAY	Bit 14 indicates which TLB associativity set should be replaced 0 Set 0 1 Set 1
15	S/L	Store/load protection instruction 0 Load miss 1 Store miss
16–31	MSR[16–31]	Copy of MSR bits 16–31

Note that in some implementations, every instruction fetch when MSR[IR] = 1 and every instruction execution requiring address translation when MSR[DR] = 1 may modify SRR1.

5.2.1.2 CSRR0 and CSRR1 Bit Settings

The e300 core also implements the CSRR0 and CSRR1 to save state for critical interrupt interrupts only. Note that the values saved in CSRR0 are the same as those saved in SRR0 for all other interrupts, and the values saved in CSRR1 are the same as those saved in the MSR for critical interrupts. CSRR0 and CSRR1 have unique SPR numbers, as described in Chapter 2, “Register Model.”

Figure 5-3 shows the format of CSRR0.

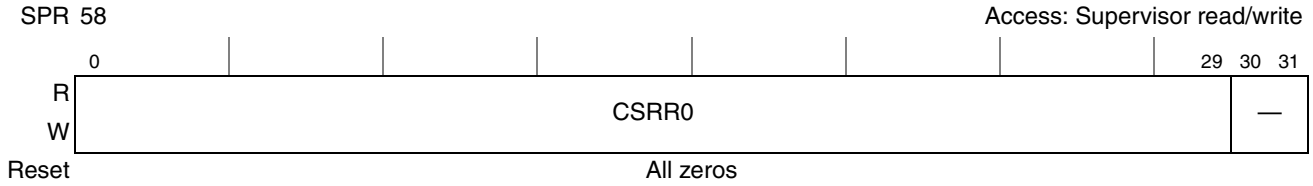


Figure 5-3. Critical Interrupt Save/Restore Register 0 (CSRR0)

When a critical interrupt occurs, CSRR0 is set to point to an instruction such that all prior instructions have completed execution and no subsequent instruction has begun execution. When an **rfci** instruction is executed, the contents of CSRR0 are copied to the next instruction address—the 32-bit address of the next instruction to be executed. Figure 5-4 shows the format of CSRR1.

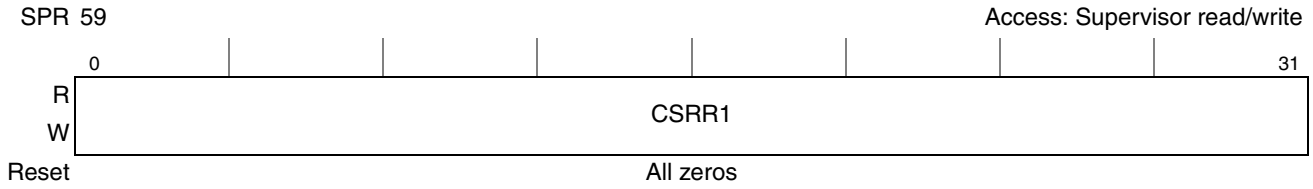


Figure 5-4. Critical Interrupt Save/Restore Register 1 (CSRR1)

When an interrupt occurs, CSRR1[0–31] are loaded with the values of MSR[0–31], which are placed in corresponding CSRR1 bit positions. When **rfci** executes, MSR[0–31] are loaded from CSRR1[0–31].

5.2.1.3 SPRG0–SPRG7

The e300 core provides eight SPRG (SPRG4–SPRG7) registers for general operating system use, such as performing a fast state save or for supporting multiprocessor implementations. SPRG0–SPRG7 have unique SPR numbers, as described in Chapter 2, "Register Model." The formats of SPRG4–SPRG7 are shown in Figure 5-5.

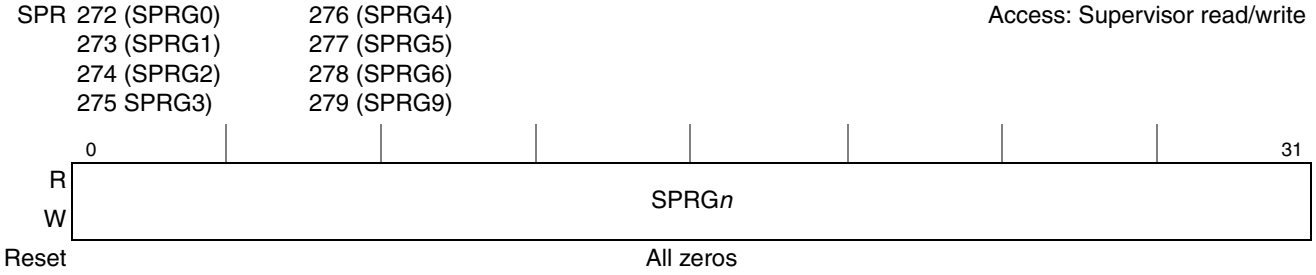


Figure 5-5. SPRG*n* Register

Table 5-7 describes conventional uses of SPRG4 –SPRG7 for the e300 core.

Table 5-7. Conventional Uses of SPRG0–SPRG7

Register	Descriptions
SPRG0	SPRG0 may be used by the operating system as needed.
SPRG1	SPRG1 may be used by the operating system as needed.
SPRG2	SPRG2may be used by the operating system as needed.
SPRG3	SPRG3 may be used by the operating system as needed.
SPRG4	Software may load a unique physical address in this register to identify an area of memory reserved for use by the first-level interrupt handler. This area must be unique for each processor in the system.
SPRG5	SPRG5 may be used as a scratch register by the first-level interrupt handler to save the content of a GPR. That GPR then can be loaded from SPRG4 and used as a base register to save other GPRs to memory.
SPRG6	SPRG6 may be used by the operating system as needed.
SPRG7	SPRG7 may be used by the operating system as needed.

5.2.1.4 MSR Bit Settings

The MSR is shown in Figure 5-6. When an interrupt occurs, MSR bits, as described in Table 5-8, are altered as determined by the interrupt.

Access: Supervisor read/write

	0	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	—	POW	TGPR	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	CE	IP	IR	DR	—	RI	LE
Reset	0000_0040 or 0000_0000 or 0001_0041 or 0001_0001																				

Figure 5-6. Machine State Register (MSR)

Table 5-8 shows the bit definitions for the MSR. Full function reserved bits are saved in SRR1 when an interrupt occurs; partial function reserved bits are not saved.

Table 5-8. MSR Bit Settings

Bits	Name	Description
0	—	Reserved. Full function.
1–4	—	Reserved. Partial function.
5–9	—	Reserved. Full function.
10–12	—	Reserved. Partial function.
13	POW	Power management enable (implementation-specific) 0 Disables programmable power modes (normal operation mode) 1 Enables programmable power modes (nap, doze, or sleep mode) This bit controls the programmable power modes only; it has no effect on dynamic power management (DPM). MSR[POW] may be altered with an **mtmsr** instruction only. Also, when altering the POW bit, software may alter only this bit in the MSR and no others. The **mtmsr** instruction must be followed by a context-synchronizing instruction. See Chapter 9, "Power Management," for more information.
14	TGPR	Temporary GPR remapping (implementation-specific) 0 Normal operation 1 TGPR mode. GPR0–GPR3 are remapped to TGPR0–TGPR3 for use by TLB miss routines The contents of GPR0–GPR3 remain unchanged while MSR[TGPR] = 1. Attempts to use GPR4–GPR31 with MSR[TGPR] = 1 yield undefined results. Temporarily replaces TGPR0–TGPR3 with GPR0–GPR3 for use by TLB miss routines. The TGPR bit is set when either an instruction TLB miss, data read miss, or data write miss interrupt is taken. The TGPR bit is cleared by an **rfi** instruction.
15	ILE	Interrupt little-endian mode. When an interrupt occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the interrupt.
16	EE	External interrupt enable 0 The processor ignores external interrupts, system management interrupts, and decrementer interrupts. 1 The processor is enabled to take an external interrupt, system management interrupt, or decrementer interrupt.
17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions (supervisor mode). 1 The processor can only execute user-level instructions (user mode).

Table 5-8. MSR Bit Settings (continued)

Bits	Name	Description
18	FP	Floating-point available (this bit is read-only on the e300c2 core) 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. If execution of one of these types of instructions is attempted, a floating-point unavailable interrupt occurs. 1 The processor can execute floating-point instructions, and can take floating-point enabled type program interrupts.
19	ME	Machine check enable 0 Machine check interrupts are disabled 1 Machine check interrupts are enabled
20	FE0	Floating-point interrupt mode 0 (see Table 5-9) (this bit is read-only on the e300c2 core)
21	SE	Single-step trace enable 0 The processor executes instructions normally 1 The processor generates a trace interrupt on the successful completion of the next instruction
22	BE	Branch trace enable 0 The processor executes branch instructions normally 1 The processor generates a trace interrupt upon the successful completion of a branch instruction
23	FE1	Floating-point interrupt mode 1 (see Table 5-9) (this bit is read-only on the e300c2 core)
24	CE	Critical interrupt enable 0 Critical interrupts disabled 1 Critical interrupts enabled; critical interrupt and **rfci** instruction enabled. The critical interrupt is an asynchronous implementation-specific interrupt. The critical = interrupt vector offset is 0x00A00. The Return From Critical Interrupt (**rfci**) instruction is implemented to return from these interrupts. Also, CSRR0 and CSRR1, are used to save and restore the processor state for critical interrupts.
25	IP	Interrupt prefix. Specifies whether an interrupt vector offset is prepended with Fs or 0s. In the following description, *nnnnn* is the offset of the interrupt. See Table 5-2. 0 Interrupts are vectored to the physical address 0x000*n_nnnn* 1 Interrupts are vectored to the physical address 0xFFF*n_nnnn*
26	IR	Instruction address translation 0 Instruction address translation is disabled 1 Instruction address translation is enabled See Chapter 6, “Memory Management.”
27	DR	Data address translation 0 Data address translation is disabled 1 Data address translation is enabled See Chapter 6, “Memory Management.”
28–29	—	Reserved. Full function. Bit 29 reserved on e300c1 and e300c2 only.
29	PMM	Performance monitor mark bit (e300c3 only). System software can set PMM when a marked process is running to enable statistics to be gathered only during the execution of the marked process. MSR[PR] and MSR[PMM] together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified in the PMLCa*n*, the state for which monitoring is enabled, counting is enabled.

Table 5-8. MSR Bit Settings (continued)

Bits	Name	Description
30	RI	Recoverable interrupt (for system reset and machine check interrupts) 0 Interrupt is not recoverable 1 Interrupt is recoverable
31	LE	Little-endian mode enable 0 The processor runs in big-endian mode 1 The processor runs in little-endian mode. For the e300 core, see Section 3.1.2, "Endian Modes and Byte Ordering," for a definition of the core operating in true little-endian mode.

The IEEE floating-point exception mode bits (FE0 and FE1) together define whether floating-point exceptions are handled precisely, imprecisely, or if they are taken at all. The possible settings and default conditions for the e300 core are shown in Table 5-9. For further details, see Chapter 6, "Interrupts," in the *Programming Environments Manual*.

Table 5-9. IEEE Floating-Point Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions disabled
0	1	Floating-point imprecise nonrecoverable[1]
1	0	Floating-point imprecise recoverable [1]
1	1	Floating-point precise mode

[1] Not implemented in the e300 core.

MSR bits are guaranteed to be written to SRR1 when the first instruction of the interrupt handler is encountered.

5.2.2 Enabling and Disabling Exceptions and Interrupts

When an exception condition exists that may cause an interrupt to be generated, it must be determined whether the interrupt is enabled for that condition as follows:

- IEEE floating-point enabled exceptions (which can generate a program interrupt) are ignored when both MSR[FE0] and MSR[FE1] are cleared. If either of these bits are set, all IEEE-enabled floating-point exceptions are taken and cause a program interrupt.
- Asynchronous, maskable interrupts (that is, the external, system management, and decrementer interrupts) are enabled by setting the MSR[EE] bit. When MSR[EE] = 0, recognition of these interrupt conditions is delayed. MSR[EE] is cleared automatically when an interrupt is taken, to delay recognition of conditions causing those interrupts.
- A machine check interrupt can occur only if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs. Individual machine check exceptions can be enabled and disabled through bits in the HID0 register, as described in Table 2-5.
- The e300 core enables the critical interrupt with the MSR[CE] bit.
- System reset interrupts cannot be masked.

5.2.3 Steps for Interrupt Processing

After it is determined that the interrupt can be taken (by confirming that any instruction-caused interrupts occurring earlier in the instruction stream have been handled, and by confirming that the interrupt is enabled for the exception condition), the processor does the following:

1. The machine status save/restore register 0 (SRR0) is loaded with an instruction address that depends on the type of interrupt. See the individual interrupt description for details about how this register is used for specific interrupts.
2. SRR1[1–4, 10–15] are loaded with information specific to the interrupt type.
3. SRR1[5–9, 16–31] are loaded with a copy of the corresponding bits of the MSR.
4. The MSR is set as described in Table 5-8. The new values take effect beginning with the fetching of the first instruction of the interrupt-handler routine located at the interrupt vector address.

   Note that MSR[IR] and MSR[DR] are cleared for all interrupt types; therefore, address translation is disabled for both instruction fetches and data accesses beginning with the first instruction of the interrupt-handler routine.
5. Instruction fetch and execution resumes, using the new MSR value, at a location specific to the interrupt type. The location is determined by adding the interrupt's vector (see Table 5-2) to the base address determined by MSR[IP]. If IP is cleared, interrupts are vectored to the physical address 0x000*n_nnnn*. If IP is set, interrupts are vectored to the physical address 0xFFF*n_nnnn*. For a machine check interrupt that occurs when MSR[ME] = 0 (machine check interrupts are disabled), the processor enters the checkstop state (the machine stops executing instructions). See Section 5.5.2, “Machine Check Interrupt (0x00200).”

Note that the same steps occur when a critical interrupt occurs (and is enabled) for the e300 core, except that CSRR0 is set instead of SRR0 and CSRR1 is set instead of SRR1.

5.2.4 Setting MSR[RI]

The operating system should handle MSR[RI] as follows:

- In the machine check and system reset interrupts—If SRR1[RI] is cleared, the interrupt is not recoverable. If it is set, the interrupt is recoverable with respect to the processor.
- In each interrupt handler—When enough state information has been saved that a machine check or system reset interrupt can reconstruct the previous state, set MSR[RI].
- In each interrupt handler—Clear MSR[RI], set the SRR0 and SRR1 (or CSRR0 and CSRR1) registers appropriately, and then execute **rfi (**or **rfci)**.
- Note that the RI bit being set indicates that, with respect to the processor, enough processor state data is valid for the processor to continue, but it does not guarantee that the interrupted process can resume.

5.2.5 Returning from an Interrupt with rfi

The Return From Interrupt (**rfi**) instruction performs context synchronization by allowing previously issued instructions to complete before returning to the interrupted process. In general, execution of the **rfi** instruction ensures the following:

- All previous instructions have completed to a point where they can no longer cause an interrupt.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The **rfi** instruction copies SRR1 bits back into the MSR.
- The instructions following this instruction execute in the context established by this instruction.

For a complete description of context synchronization, refer to Chapter 6, “Interrupts,” in the *Programming Environments Manual.*

5.2.6 Returning from an Interrupt with rfci

The Return From Critical Interrupt (**rfci**) is a e300 core-only supervisor level instruction that performs context synchronization by allowing previously issued instructions to complete before returning to the interrupted process. The **rfci** instruction performs the same functions as **rfi**, except that it uses CSRR0 and CSRR1 to restore the processor state. Thus, execution of the **rfci** instruction ensures the following:

- CSRR1[0, 5–9, 16–31] are placed into the corresponding bits of the MSR. If the new MSR value does not enable any pending interrupts, the next instruction is fetched from the address defined by CSRR0[0–29] || 0b00.
- If the new MSR value enables one or more pending interrupts, the interrupt associated with the highest priority pending interrupt is generated. In this case, the interrupt processing mechanism places in SRR0 the address of the instruction which would have executed next had the interrupt not occurred.

5.3 Process Switching

The operating system should execute one of the following when processes are switched:

- The **sync** instruction, which orders the effects of instruction execution. All instructions previously initiated appear to have completed before the **sync** instruction completes, and no subsequent instructions appear to be initiated until the **sync** instruction completes. For an example showing the use of a **sync** instruction, see Chapter 2, “Register Set,” of the *Programming Environments Manual.*
- The **isync** instruction, which waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, protection, etc.) established by the previous instructions.
- The **stwcx.** instruction, to clear any outstanding reservations, which ensures that an **lwarx** instruction in the old process is not paired with an **stwcx.** instruction in the new process.

The operating system should set the MSR[RI] bit as described in Section 5.2.4, “Setting MSR[RI].”

5.4 Interrupt Latencies

Latencies for taking various interrupts depend on the state of the machine when the exception conditions occur. This latency may be as short as one cycle, in which case an interrupt is signaled in the cycle following the appearance of the exception condition. The latencies are as follows:

- Hard reset and machine check—In most cases, a hard reset or machine check interrupt will have a single-cycle latency. A two- to three-cycle delay may occur only when a predicted instruction is next to complete, and the branch guess that forced this instruction to be predicted was resolved to be incorrect.
- Soft reset—The latency of a soft reset interrupt is affected by recoverability. The time to reach a recoverable state may depend on the time needed to complete or except an instruction at the point of completion, the time needed to drain the completed store queue (see Section 7.1, "Terminology and Conventions," for the definition), and the time waiting for a correct empty state so that a valid MSR[IP] may be saved. For lower-priority externally-generated interrupts, a delay may be incurred waiting for another interrupt generated while reaching a recoverable state to be serviced.

Further delays are possible for other types of interrupts depending on the number and type of instructions that must be completed before those interrupts may be serviced. See Section 5.1.2, "Summary of Front-End Interrupt Handling," to determine possible maximum latencies for different interrupts.

5.5 Interrupt Definitions

Table 5-10 shows all the types of interrupts that can occur with the e300 core and the MSR bit settings when the processor transitions to supervisor mode. The state of these bits prior to the interrupt is typically stored in SRR1 (or CSRR1 for critical interrupts on the e300 core). Note that MSR[CE] is cleared for the system reset, machine check, and critical interrupts.

Table 5-10. MSR Setting Due to Interrupt

Interrupt Type	MSR Bit																
	POW	TGPR	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	CE[1]	IP	IR	DR	RI	LE
System reset	0	0	—	0	0	0	—	0	0	0	0	0	1	0	0	0	ILE
Machine check	0	0	—	0	0	0	0	0	0	0	0	0	—	0	0	0	ILE
DSI	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
ISI	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
External	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
Alignment	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
Program	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
Floating-point unavailable	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
Decrementer	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
Critical Interrupt	0	0	—	0	0	0	—	0	0	0	0	0	—	0	0	0	ILE
System call	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
Trace	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
ITLB miss	0	1	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
DTLB miss on load	0	1	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
DTLB miss on store	0	1	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE
Instruction address breakpoint	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE

Table 5-10. MSR Setting Due to Interrupt (continued)

Interrupt Type	MSR Bit																
	POW	TGPR	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	CE[1]	IP	IR	DR	RI	LE
System management interrupt	0	0	—	0	0	0	—	0	0	0	0	—	—	0	0	0	ILE

Note:
0 Bit is cleared.
1 Bit is set.
ILE Bit is copied from the ILE bit in the MSR.
— Bit is not altered.
Reserved bits are read as if written as 0.

[1] e300 core only.

5.5.1 Reset Interrupts (0x00100)

The system reset interrupt is a nonmaskable, asynchronous interrupt signaled to the e300 core either through the assertion of the reset signals (*sreset* or *hreset*). The assertion of the soft reset signal, *sreset*, causes the system reset interrupt to be taken and the physical base address of the handler is determined by the MSR[IP] bit.

The assertion of the hard reset signal, *hreset*, causes the system reset interrupt to be taken.

Note that there are some byte ordering precautions necessary when coming out of reset in big-endian mode and switching to little-endian mode. The following sections describe the differences between a hard and soft reset and the byte ordering implications for reset interrupt handling.

5.5.1.1 Hard Reset and Power-On Reset

As described in Section 5.1.2, “Summary of Front-End Interrupt Handling,” the hard reset interrupt is a nonrecoverable, nonmaskable asynchronous interrupt. When *hreset* is asserted or at power-on reset (POR), the e300 core immediately branches to the address determined by the state of the *msrip* signal, as described in Table 5-11, without attempting to reach a recoverable state.

Table 5-11. Hard Reset MSR Value and Interrupt Vector

msrip	MSR[0–31]	Fetch Instructions from Handler at System Reset Vector
Asserted	0x0000_0040 (MSR[IP] = 1)	0xFFF0_0100
Negated	0x0000_0000 (MSR[IP] = 0)	0x0000_0100

A hard reset has the highest priority of any interrupt, and is always nonrecoverable. Table 5-12 shows the state of the machine just before it fetches the first instruction of the system reset handler after a hard reset.

Table 5-12. Settings Caused by Hard Reset

Register	Setting	Register	Setting
GPRs	Unknown	PVR	See Table 2-2
FPRs	Unknown	HID0	0000_0000
FPSCR	00000000	HID1	0000_0000
CR	All 0s	HID2	0000_0000 or 0800_0000
SRs	Unknown	DMISS and IMISS	All 0s
MSR	0000_0040 or 0000_0000 or 0001_0041 or 0001_0001	DCMP and ICMP	All 0s
XER	0000_0000	RPA	All 0s
TBU	0000_0000	IABR	All 0s
TBL	0000_0000	DSISR	0000_0000
LR	0000_0000	DAR	0000_0000
CTR	0000_0000	DEC	FFFF_FFFF
SDR1	0000_0000	HASH1	0000_0000
SRR0 (and CSRR0)	0000_0000	HASH2	0000_0000
SRR1 (and CSRR1)	0000_0000	TLBs	Unknown
SPRGs	0000_0000	Cache	All cache blocks invalidated
Tag directory	All 0s. (However, LRU bits are initialized so each side of the cache has a unique LRU value.)	BATs	Unknown

The *hreset* signal can be asserted for the following reasons:

- System power-on reset
- System reset from a panel switch

The following is also true after a hard reset operation:

- External checkstops are enabled
- The on-chip test interface has given control of the I/Os to the rest of the chip for functional use
- Because the reset interrupt has data and instruction translation disabled (MSR[DR] and MSR[IR] both cleared), the chip operates in real addressing mode as described in Section 6.2, “Real Addressing Mode.”

5.5.1.2 Soft Reset

As described in Section 5.1.2, “Summary of Front-End Interrupt Handling,” the soft reset interrupt is a type of system reset interrupt that is recoverable, nonmaskable, and asynchronous. When *sreset* is asserted, the processor attempts to reach a recoverable state by allowing the next instruction to either complete or cause an interrupt, blocking the completion of subsequent instructions, and allowing the completed store queue to drain (see Section 7.1, “Terminology and Conventions,” for the definition).

Unlike a hard reset, no registers or latches are initialized; however, the instruction cache is disabled (HID0[ICE] = 0). After *sreset* is recognized as asserted, the processor begins fetching instructions from the system reset routine at offset 0x0100. When a soft reset occurs, registers are set as shown in Table 5-13. A soft reset is recoverable provided that attaining the recoverable state does not cause a machine check interrupt. This interrupt case is third in priority, following hard reset and machine check.

When a soft reset occurs, registers are set as shown in Table 5-13 in addition to the clearing of HID0[ICE].

Table 5-13. Soft Reset Interrupt—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that the processor would have attempted to complete next if no interrupt conditions were present.
SRR1	0–15 Cleared 16–31 Loaded from MSR[16–31]. Note that if the processor state is corrupted to the extent that execution cannot be reliably restarted, SRR1[30] is cleared.
MSR	POW 0 FP 0 FE1 0 RI 0 TGPR 0 ME — CE 0 LE Set to value of ILE ILE — FE0 0 IP — EE 0 SE 0 IR 0 PR 0 BE 0 DR 0

5.5.1.3 Byte Ordering Considerations

All interrupt routines are executed in the endian mode determined by the setting of the MSR[ILE], MSR[LE], and HID2[LET] bits (see Table 1-2 for endian mode indication) when the interrupt is taken. A special case is the system reset interrupt for both hard and soft reset for the e300 core. When the *tle* signal is negated at the time *hreset* is negated, the system interrupt handler of the device enters into big-endian mode. If MSR[ILE], MSR[LE], and HID2[LET] are subsequently set (during or after the reset routine has completed), a subsequent soft reset causes the system reset interrupt handler to be entered in true little-endian mode, potentially resulting in illegal instruction execution (if the beginning of the handler is written assuming big-endian code). Note that the reverse occurs for true little-endian mode.

The following assembly language code highlights register settings necessary when in big-endian mode coming out of hard reset and subsequently changing the processor state to true little-endian mode and setting the MSR[ILE], MSR[LE], and HID2[LET] bits. The first eight instructions of the system reset interrupt handler is written in big-endian format, in order to facilitate the mode switch. The rest of the reset handler is written in true little-endian format for the remaining supervisor or OS code. This reset code assumes that caching is not enabled out of reset. Due to the complexities involved with keeping the memory system coherent, it is strongly recommended not to change endianess at any other time once it is determined at hard reset.

```
.orig 0xFFF0 0100 # default IP vector
# Begin HRESET_ handler with Big-Endian Mode
xor      r2,r2,r2          initialize register
xor      r1,r1,r1         # initialize register
oris     r2,r2,0x0800     # set bit in r2 for HID2[4]LET
mtspr    HID2,r2          # load HID2 setting LET bit
oris     r1,r1,0x0001     # set bit in r1 for MSR[15]ILE
ori      r1,r1,0x0001     # set bit in r1 for MSR[31]LE
```

```
mtmsr    r1               # load MSR setting ILE and LE bits
isync                     # wait for all instructions to complete

# End Big-Endian mode, True Little-Endian enabled
# modify the 8 Big-Endian instructions into valid True Little-Endian instructions
# True Little-Endian Mode
mtspr    SRR1,r1          # load the Machine State with LE enabled
xor      r0,r0,r0         # initialize register
oris     r0,r0,0x0001     # set Starting address at b'0001 0000
mtspr    SRR0,r0          # load the next instruction address
# whatever instructions the supervisor/OS wants.
rfi                       # return from HRESET_ interrupt routine
# End HRESET_ handler in True Little-Endian Mode
```

See Section 3.1.2, "Endian Modes and Byte Ordering," for more information on the endian modes of the e300 core.

5.5.2 Machine Check Interrupt (0x00200)

The e300 core conditionally initiates a machine check interrupt after detecting the assertion of the $\overline{tea}$ or $\overline{mcp}$ signals on the coherent system bus (CSB) (assuming the machine check is enabled with MSR[ME] = 1). The assertion of one of these signals indicates that a bus error occurred and the system terminates the current transaction. One clock cycle after the signal is asserted, the data bus signals go to the high-impedance state; however, data entering the GPR or the cache is not invalidated. Note that if HID0[EMCP] is cleared, the core ignores the assertion of the $\overline{mcp}$ signal but continues to monitor the $\overline{tea}$ signal.

A machine check interrupt also occurs when an address or data parity error is detected on the bus and the address or data parity error is enabled in HID0. See Section 2.2.1, "Hardware Implementation Register 0 (HID0)," for more information.

Note that the e300 core makes no attempt to force recoverability on a machine check; however, it does guarantee that the machine check interrupt is always taken immediately upon request, with a nonpredicted address saved in SRR0, regardless of the current machine state. Because pending stores in the store queue (see Figure 7-6) are not canceled when a machine check interrupt occurs, two consecutive stores that result in the assertion of $\overline{tea}$ can cause the processor to checkstop. To prevent a checkstop in this case, a **sync** instruction must be placed between two stores that can result in assertion of $\overline{tea}$.

Software can use the machine check interrupt handler in a recoverable mode to probe memory. For this case, a **sync**, load, **sync** instruction sequence is used. If the load access results in a system error (for example, the assertion of $\overline{tea}$), the processor can handle this in a recoverable state. If the **sync** instruction is not used, a second access to the same address as the first load could cause the processor to enter the checkstop state.

If the MSR[ME] bit is set, the interrupt is recognized and handled; otherwise, the e300 core attempts to enter an internal checkstop. Note that the resulting machine check interrupt has priority over any interrupts caused by the instruction that generated the bus operation.

Machine check interrupts are only enabled when MSR[ME] = 1; this is described in Section 5.5.2.1, "Machine Check Interrupt Enabled (MSR[ME] = 1)." If MSR[ME] = 0 and a machine check occurs, the

processor enters the checkstop state; this is described in Section 5.5.2.2, "Checkstop State (MSR[ME] = 0)."

5.5.2.1 Machine Check Interrupt Enabled (MSR[ME] = 1)

When a machine check interrupt is taken, registers are updated as shown in Table 5-14.

When a machine check interrupt is taken, instruction execution for the handler begins at offset 0x00200 from the physical base address indicated by MSR[IP].

In order to return to the main program, the interrupt handler should do the following:

1. SRR0 and SRR1 should be given the values to be used by the **rfi** instruction
2. Execute **rfi**

Table 5-14. Machine Check Interrupt—Register Settings

Register	Setting Description
SRR0	Set to the address of the next instruction that would have been completed in the interrupted instruction stream. Neither this instruction nor any others beyond it will have been completed. All preceding instructions will have been completed.
SRR1	0–9 Cleared 10 Instruction cache parity error caused interrupt 11 Data cache parity error caused interrupt 12 *mcp*—Machine check signal caused interrupt 13 *tea*—Transfer error acknowledge signal caused interrupt 14 *dpe*—Data parity error condition (and signal assertion) caused interrupt 15 *ape*—Address parity error condition (and signal assertion) caused interrupt 16–29 Loaded from MSR[16–29] 30 0 for instruction cache parity error, data cache parity error, *tea*, *dpe*, *ape*; loaded from MSR[30] for *mcp*. If *mcp* and *tea* are asserted simultaneously, then SRR1[30] is cleared and the interrupt is not recoverable. 31 Loaded from MSR[31]
MSR	POW 0 FP 0 FE1 0 RI 0 TGPR 0 ME — CE 0 LE Set to value of ILE ILE — FE0 0 IP — EE 0 SE 0 IR 0 PR 0 BE 0 DR 0

Note: When a machine check interrupt is taken, the interrupt should set MSR[ME] as soon as it is practical to handle another *tea* assertion. Otherwise, subsequent *tea* assertions cause the processor to automatically enter the checkstop state.

5.5.2.2 Checkstop State (MSR[ME] = 0)

When the e300 core enters the checkstop state, it asserts the checkstop output signal, *ckstp_out*. The following events cause the e300 core to enter the checkstop state:

- Machine check interrupt occurs with MSR[ME] cleared
- External checkstop input, *ckstp_in*, is asserted.

When a processor is in the checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the processor. The contents of all latches are frozen within two cycles upon

entering the checkstop state so that the state of the processor can be analyzed as an aid in problem determination.

Note that not all processors that implement the PowerPC architecture provide the same level of error checking. The reasons a processor can enter checkstop state are implementation-dependent.

5.5.3 DSI Interrupt (0x00300)

A DSI interrupt occurs when no higher priority interrupt exists and a data memory access cannot be performed. The condition that caused the DSI interrupt can be determined by reading the DSISR register, a supervisor-level SPR (SPR18) that can be read by using the **mfspr** instruction. Bit settings are provided in Table 5-15. Table 5-15 also indicates the memory element that is saved to the DAR.

Table 5-15. DSI Interrupt—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that caused the interrupt.
SRR1	0–15 Cleared 16–31 Loaded with MSR[16–31]
MSR	POW 0 FP 0 FE1 0 RI 0 TGPR 0 ME — CE — LE Set to value of ILE ILE — FE0 0 IP — EE 0 SE 0 IR 0 PR 0 BE 0 DR 0
DSISR	0 Cleared 1 Set by the data TLB miss interrupt the translation of an attempted access is not found in the primary hash table entry group (HTEG), or in the rehashed secondary HTEG, or in the range of a DBAT register; otherwise cleared. 2–3 Cleared 4 Set if a memory access is not permitted by the page or BAT protection mechanism; otherwise cleared. 5 Cleared 6 Set for a store operation and cleared for a load operation 9 Set if a data address breakpoint interrupt occurs when the data (bit 29) in the DABR or DABR2 matches the next data access (load or store instruction) to complete in the completion unit. The different breakpoints are enabled as follows: • Write breakpoints enabled when DABR[30] is set • Read breakpoints enabled when DABR[31] is set 7–31 Cleared
DAR	Set to the effective address of a memory element as described in the following list: • A byte in the first word accessed in the page that caused the DSI interrupt, for a byte, half word, or word memory access. • A byte in the first word accessed in the BAT area that caused the DSI interrupt for a byte, half word, or word access to a BAT area. • A byte in the block that caused the interrupt for **icbi**, **dcbz**, **dcbst**, **dcbf**, or **dcbi** instructions. • The EA that causes a data breakpoint.

DSI interrupts can occur for any of the following reasons:

- The instruction is not supported for the type of memory addressed
- The attempted access violates the memory protection defined by SR[Ks,Kp], PTE[PP], or DBATn[PP].

Note that the OEA specifies an additional case that may cause a DSI interrupt—when an effective address for a load, store, or cache operation cannot be translated by the TLBs. On the e300 core, this condition causes a TLB miss interrupt instead. These scenarios are common among all processors that implement the PowerPC architecture.

Finally, the e300 core causes a DSI interrupt when either DABR or DABR2 is enabled and the address of an access matches with the value in the CEA field and the breakpoint is enabled for the type of access (read or write) in DABR/DABR2. See Chapter 10, "Debug Features," for more information.

DSI interrupts can be generated by load/store instructions and cache control instructions (**dcbi**, **dcbz**, **dcbst**, and **dcbf**).

The e300 core supports the crossing of page boundaries. However, if the second page has a translation error or protection violation associated with it, the e300 core takes the DSI interrupt in the middle of the instruction. In this case, the instruction is re-executed.

If an **stwcx.** instruction has an effective address for which a normal store operation would cause a DSI interrupt, the e300 core takes the DSI interrupt without checking for the reservation.

If the XER indicates that the byte count for an **lswi** or **stswi** instruction is zero, a DSI interrupt does not occur, regardless of the effective address.

The condition that caused the interrupt is defined in the DSISR. These conditions also use the data address register (DAR) as shown in Table 5-15.

When a DSI interrupt is taken, instruction execution for the handler begins at offset 0x00300 from the physical base address indicated by MSR[IP].

The architecture permits certain instructions to be partially executed when they cause a DSI interrupt. These are as follows:

- Load string instructions—some registers in the range of registers to be loaded may have been loaded.
- Store multiple or store string instructions—some bytes of memory in the range addressed may have been updated.

In these cases, the number of registers and amount of memory altered are instruction- and boundary-dependent. However, memory protection is not violated.

For update forms, the update register (**r**A) is not altered.

5.5.4 ISI Interrupt (0x00400)

The ISI interrupt is implemented as it is defined by the PowerPC architecture. An ISI interrupt occurs when no higher priority interrupt exists and an attempt to fetch the next instruction fails for any of the following reasons:

- If an instruction TLB miss fails to find the desired PTE, then a page fault is synthesized. The ITLB miss handler branches to the ISI interrupt to retrieve the translation from a storage device.
- An attempt is made to fetch an instruction from no-execute memory
- The fetch access violates memory protection

Register settings for this interrupt are described in Chapter 6, “Interrupts,” in the *Programming Environments Manual*.

When an ISI interrupt is taken, instruction execution for the handler begins at offset 0x00400 from the physical base address indicated by MSR[IP].

5.5.5 External Interrupt (0x00500)

An external interrupt is signaled to the e300 core by the assertion of the $\overline{int}$ signal as described in Section 8.3.1, “External Interrupts.” The interrupt may not be recognized if a higher priority interrupt occurs simultaneously or if the MSR[EE] bit is cleared when $\overline{int}$ is asserted.

After the $\overline{int}$ is recognized, the e300 core generates a recoverable halt to instruction completion. The e300 core allows the next instruction in program order to complete, including handling any interrupts that instruction may generate. However, the e300 core blocks subsequent instructions from completing and allows any outstanding stores to occur to system memory. If any other interrupts are encountered in this process, they are taken first and the external interrupt is delayed until a recoverable halt is achieved. At this time, the e300 core saves the state information and takes the external interrupt as defined by the PowerPC architecture.

The register settings for the external interrupt are shown in Table 5-16.

Table 5-16. External Interrupt—Register Settings

Register	Setting			
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.			
SRR1	0–15 Cleared 16–31 Loaded from MSR[16–31]			
MSR	POW 0 TGPR 0 ILE — EE 0 PR 0	FP 0 ME — FE0 0 SE 0 BE 0	FE1 0 CE — IP — IR 0 DR 0	RI 0 LE Set to value of ILE

When an external interrupt is taken, instruction execution for the handler begins at offset 0x00500 from the physical base address indicated by MSR[IP].

The e300 core only recognizes the interrupt condition ($\overline{int}$ asserted) if the MSR[EE] bit is set; it ignores the interrupt condition if the MSR[EE] bit is cleared. To guarantee that the external interrupt is taken, the $\overline{int}$ signal must be held asserted until the e300 core takes the interrupt. If the $\overline{int}$ signal is negated before the interrupt is taken, the e300 core is not guaranteed to take an external interrupt. The interrupt handler must send a command to the device that asserted $\overline{int}$, acknowledging the interrupt and instructing the device to negate $\overline{int}$ before the handler re-enables recognition of external interrupts.

5.5.6 Alignment Interrupt (0x00600)

This section describes conditions that can cause alignment interrupts in the e300 core. The e300 core implements the alignment interrupt as it is defined in the PowerPC architecture. For information on bit

settings and how interrupt conditions are detected, refer to the *Programming Environments Manual*. Note that the PowerPC architecture allows individual processors to determine whether an interrupt is required to handle various alignment conditions.

Similar to DSI interrupts, alignment interrupts use the SRR0 and SRR1 to save the machine state and the DSISR to determine the source of the interrupt. The e300 core initiates an alignment interrupt when it detects any of the following conditions:

- The operand of a floating-point load or store operation is not word-aligned
- The operand of an **lmw**, **stmw**, **lwarx**, or **stwcx.** instruction is not word-aligned.
- A multiple or string access is attempted with the MSR[LE] bit set
- The operand of a **dcbz** instruction is in a page that is write-through or caching-inhibited
- The instruction is **lswi**, **lswx**, **stswi**, **stswx**, and the core is in little-endian mode. This applies to both PowerPC little-endian in previous cores and true little-endian mode for the e300 core. Note that the e300 core does not support PowerPC (modified) little-endian mode.

Note that the e300 core does not generate an alignment interrupt for a misaligned little-endian access (MSR[LE] = 1).

The register settings for alignment interrupts are shown in Table 5-16.

The architecture does not support the use of a misaligned EA by **lwarx** or **stwcx.** instructions. If one of these instructions specifies a misaligned EA, the interrupt handler should not emulate the instruction, but should treat the occurrence as a programming error.

Table 5-17. Alignment Interrupt—Register Settings

Register	Setting			
SRR0	Set to the effective address of the instruction that caused the interrupt			
SRR1	0–15 Cleared 16–31 Loaded from MSR[16–31]			
MSR	POW 0 TGPR 0 ILE — EE 0 PR 0	FP 0 ME — FE0 0 SE 0 BE 0	FE1 0 CE — IP — IR 0 DR 0	RI 0 LE Set to value of ILE

Table 5-17. Alignment Interrupt—Register Settings (continued)

Register	Setting
DSISR	0–11 Cleared 12–13 Cleared. (Note that these bits can be set by several 64-bit PowerPC instructions that are not supported in the e300 core.) 14 Cleared 15–16 For instructions that use register indirect with index addressing—set to bits 29–30 of the instruction For instructions that use register indirect with immediate index addressing—cleared 17 For instructions that use register indirect with index addressing—set to bit 25 of the instruction For instructions that use register indirect with immediate index addressing—set to bit 5 of the instruction 18–21 For instructions that use register indirect with index addressing—set to bits 21–24 of the instruction For instructions that use register indirect with immediate index addressing—set to bits 1–4 of the instruction 22–26 Set to bits 6–10 (identifying either the source or destination) of the instruction. Undefined for **dcbz**. 27–31 Set to bits 11–15 of the instruction (**r**A). Set to either bits 11–15 of the instruction or to any register number not in the range of registers loaded by a valid form instruction, for **lmw**, **lswi**, and **lswx** instructions. Otherwise undefined.
DAR	Set to the EA of the data access as computed by the instruction causing the alignment interrupt. When the operand of an **lmw**, **stmw**, **lwarx**, or **stwcx.** instruction is not word-aligned, that address value + 4 is stored into the DAR.

5.5.6.1 Integer Alignment Exceptions

The e300 core is optimized for load and store operations that are aligned on natural boundaries. Operations that are not naturally aligned may suffer performance degradation, depending on the type of operation, the boundaries crossed, and the mode that the processor is in during execution. More specifically, these operations may either cause an alignment interrupt or they may cause the processor to break the memory access into multiple, smaller accesses with respect to the cache and the memory subsystem.

The e300 core can initiate an alignment interrupt for the access shown in Table 5-18. In this case, the appropriate range check is performed before the instruction begins execution. As a result, if an alignment interrupt is taken, it is guaranteed that no portion of the instruction has been executed.

Table 5-18. Access Types

MSR[DR]	SR[T]	Access Type
1	0	Page-address translation access

A page-address translation access occurs when MSR[DR] is set and there is not a match in the BAT. Note the following points:

- The following is true for all loads and stores except strings/multiples (note that these four cases do not cause an alignment interrupt in the e300c2 core):
  - Byte operands never cause an alignment interrupt
  - Half-word operands can cause an alignment interrupt if the EA ends in 0xFFF
  - Word operands can cause an alignment interrupt if the EA ends in 0xFFD–FFF
  - Double-word operands cause an alignment interrupt if the EA ends in 0xFF9–FFF

- The **dcbz** instruction causes an alignment interrupt if the access is to a page or block with the W (write-through) or I (cache-inhibit) bit set in the TLB or BAT, respectively.

A misaligned memory access that does not cause an alignment interrupt do not perform as well as an aligned access of the same type. The resulting performance degradation due to misaligned accesses depends on how well each individual access behaves with respect to the memory hierarchy. At a minimum, additional cache access cycles are required that can delay other processor resources from using the cache. More dramatically, for an access to a noncacheable page, each discrete access involves individual processor bus operations that reduce the effective bandwidth of that bus.

Finally, note that when the e300 core is in page address translation mode, there is no special handling for accesses that fall into BAT regions.

5.5.6.2 Load/Store Multiple Alignment Exceptions

Most alignment interrupts store the address as computed by the instruction in the DAR. However, when the operand of an **lmw**, **stmw**, **lwarx**, or **stwcx.** instruction is not word-aligned that address value + 4 is stored into the DAR.

5.5.7 Program Interrupt (0x00700)

The e300 core implements the program interrupt as it is defined by the PowerPC architecture (OEA). A program interrupt occurs when no higher priority interrupt exists and one or more of the exception conditions defined in the OEA occur.

When a program interrupt is taken, instruction execution for the handler begins at offset 0x00700 from the physical base address indicated by MSR[IP]. The exception conditions are as follows:

- Floating-point enabled exception—These exceptions correspond to IEEE-defined exception conditions, such as overflows, and divide by zeros that may occur during the execution of a floating-point arithmetic instruction. As a group, these exceptions are enabled by the FE0 and FE1 bits in the MSR. Individual conditions are enabled by specific bits in the FPSCR. For general information about this interrupt, see the *Programming Environments Manual*. For more information about how these exceptions are implemented in the e300 core, see Section 5.5.7.1, “IEEE Floating-Point Exception Program Interrupts.”
- Illegal instruction—An illegal instruction program interrupt is generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields (including PowerPC instructions not implemented in the e300 core). These do not include those optional instructions treated as no-ops.
- Privileged instruction—A privileged instruction type program interrupt is generated when the execution of a privileged instruction is attempted and the MSR register user privilege bit, MSR[PR], is set. In the e300 core, this interrupt is generated for **mtspr** or **mfspr** with an invalid SPR field if SPR[0] = 1 and MSR[PR] = 1. This may not be true for all processors that implement the PowerPC architecture.
- Trap—A trap type program interrupt is generated when any of the conditions specified in a trap instruction is met.

5.5.7.1 IEEE Floating-Point Exception Program Interrupts

Floating-point exceptions are signaled by condition bits set in the floating-point status and control register (FPSCR). They can cause the system floating-point enabled interrupt handler to be invoked. The e300 core handles all floating-point exceptions precisely. The e300 core implements the FPSCR as it is defined by the PowerPC architecture; for more information about the FPSCR, see the *Programming Environments Manual*.

Floating-point operations that change exception sticky bits in the FPSCR may suffer a performance penalty. When an exception is disabled in the FPSCR and MSR[FE] = 0, updates to the FPSCR exception sticky bits are serialized at the completion stage. This serialization may result in a one- or two-cycle execution delay. The penalty is incurred only when the exception bit is changed and not on subsequent operations with the same exception. See Chapter 7, "Instruction Timing," for a full description of completion serialization.

When an exception is enabled in the FPSCR, the instruction traps to the emulation trap interrupt vector without updating the FPSCR or the target FPR. The emulation trap interrupt handler is required to complete the instruction. The emulation trap interrupt handler is invoked regardless of the FE setting in the MSR.

The two IEEE floating-point imprecise modes, defined by the PowerPC architecture when MSR[FE0] ≠ MSR[FE1], are treated as precise (that is, MSR[FE0] = MSR[FE1] = 1). This is regardless of the setting of MSR[NI].

For the highest and most predictable floating-point performance, all exceptions should be disabled in the FPSCR and MSR. For more information about the program exception, see the *Programming Environments Manual*.

5.5.7.2 Illegal, Reserved, and Unimplemented Instructions Program Interrupts

In accordance with the PowerPC architecture, the e300 core considers all instructions defined for 64-bit implementations and unimplemented optional instructions, such as **fsqrt**, **eciwx**, and **ecowx** as illegal and takes a program interrupt when one of these instructions is encountered. Likewise, if a supervisor-level instruction is encountered when the processor is in user-level mode, a privileged instruction-type program interrupt is taken.

5.5.8 Floating-Point Unavailable Interrupt (0x00800)

A floating-point unavailable interrupt occurs when no higher priority interrupt exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, and move instructions), and the floating-point available bit in the MSR is disabled (MSR[FP] = 0). Register settings for this interrupt are described in Chapter 6, "Interrupts," in the *Programming Environments Manual*

When a floating-point unavailable interrupt is taken, instruction execution for the handler begins at offset 0x00800 from the physical base address indicated by MSR[IP].

5.5.9 Decrementer Interrupt (0x00900)

The e300 core implements the decrementer interrupt as it is defined in the PowerPC architecture. A decrementer interrupt request is made when the decrementer counts down through zero. The request is held until there are no higher priority interrupts and MSR[EE] = 1. At this point the decrementer interrupt is taken. If multiple decrementer interrupt requests are received before the first can be reported, only one interrupt is reported. The occurrence of a decrementer interrupt cancels the request. Register settings for this interrupt are described in Chapter 6, “Interrupts,” in the *Programming Environments Manual.*

When a decrementer interrupt is taken, instruction execution for the handler begins at offset 0x00900 from the physical base address indicated by MSR[IP].

5.5.10 Critical Interrupt (0x00A00)

A critical interrupt is signaled to the e300 core by the assertion of the $\overline{int}$ signal. The interrupt may not be recognized if a higher priority interrupt occurs simultaneously or if the MSR[CE] bit is cleared when $\overline{cint}$ is asserted.

The following events occur when the e300 recognizes the assertion of $\overline{cint}$:

- Multi-cycle instructions not in the completion stage are terminated
- Outstanding load or store instructions that have not been completed are terminated
- Any outstanding page table search activity is terminated
- The effective address for resuming program execution is saved into CSRR0
- The contents of MSR are saved into CSRR1
- The MSR register is loaded with all zeros except the IP, ILE, and ME bits which remain unchanged
- Interrupt processing starts at offset value 0x00A00 from the physical base address indicated by MSR[IP]

Some types of instructions (for example load multiple/string and floating-point instructions) cause additional interrupt recognition latency. Timing critical applications must consider these instruction execution latencies in calculating worst-case interrupt recognition latency.

Upon returning from a critical interrupt routine, the core restarts any terminated or uncompleted instructions, including terminated load multiple or load string instructions. Note that these restarted load instructions may cause side-effects on peripheral devices that have auto-decrementer or status bit changes caused by the subsequent load accesses.

The register settings for the critical interrupt are shown in Table 5-16.

Table 5-19. Critical Interrupt—Register Settings

Register	Setting
CSRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.
CSRR1	0–15 Cleared 16–31 Loaded from MSR[16–31]
MSR	POW 0 FP 0 FE1 0 RI 0 TGPR 0 ME — CE 0 LE Set to value of ILE ILE — FE0 0 IP — EE 0 SE 0 IR 0 PR 0 BE 0 DR 0

The e300 core only recognizes the interrupt condition ($\overline{cint}$ asserted) if the MSR[CE] bit is set; it ignores the interrupt condition if the MSR[CE] bit is cleared. To guarantee that the critical interrupt is taken, the $\overline{cint}$ signal must be held asserted until the e300 core takes the interrupt. If the $\overline{cint}$ signal is negated before the interrupt is taken, the e300 core is not guaranteed to take a critical interrupt. The interrupt must send a command to the device that asserted $\overline{cint}$, acknowledging the interrupt and instructing the device to negate $\overline{cint}$ before the handler re-enables recognition of critical interrupts.

The additional SPRG4–7 registers on the e300 core can reduce overall latency for critical interrupts, as fewer GPRs need to be saved upon entering a critical interrupt routine. The e300 core also implements the **rfci** instruction for specifically returning from critical interrupt routines and restoring the processor state from CSRR0 and CSRR1.

5.5.11 System Call Interrupt (0x00C00)

The e300 core implements the system call interrupt as it is defined by the PowerPC architecture. A system call interrupt request is made when a system call (**sc**) instruction is completed. If no higher priority interrupt exists, the system call interrupt is taken, with SRR0 being set to the EA of the instruction following the **sc** instruction. Register settings for this interrupt are described in Chapter 6, “Interrupts,” in the *Programming Environments Manual.*

When a system call interrupt is taken, instruction execution for the handler begins at offset 0x00C00 from the physical base address indicated by MSR[IP].

5.5.12 Trace Interrupt (0x00D00)

The trace interrupt is taken under one of the following conditions:

- When MSR[SE] is set, a single-step instruction trace interrupt is taken when no higher priority interrupt exists and any instruction (other than **rfi, rfci, mtmsr,** or **isync**) is successfully completed. Note that other processors will take the trace interrupt on **isync** instructions (when MSR[SE] is set); the e300 core does not take the trace interrupt on **isync** instructions. Single-step instruction trace mode is described in Section 5.5.12.1, “Single-Step Instruction Trace Mode.”

- When MSR[BE] is set, the branch trace interrupt is taken after each branch instruction is completed.
- The e300 core deviates from the architecture by not taking trace interrupts on **isync** instructions. Single-step instruction trace mode is described in Section 5.5.12.2, “Branch Trace Mode.”

Successful completion implies that the instruction caused no other interrupts. A trace interrupt is never taken for an **sc** or trap instruction that takes a trap interrupt.

MSR[SE] and MSR[BE] are cleared when the trace interrupt is taken. In the normal use of this function, MSR[SE] and MSR[BE] are restored when the interrupt handler returns to the interrupted program using an **rfi** instruction.

Register settings for the trace mode are described in Table 5-20.

Table 5-20. Trace Interrupt—Register Settings

Register	Setting Description			
SRR0	Set to the address of the instruction following the one for which the trace interrupt was generated.			
SRR1	0–15 Cleared 16–31 Loaded from MSR[16–31]			
MSR	POW 0 TGPR 0 ILE — EE 0 PR 0	FP 0 ME — FE0 0 SE 0 BE 0	FE1 0 CE — IP — IR 0 DR 0	RI 0 LE Set to value of ILE

If trace and breakpoint conditions occur simultaneously, the breakpoint conditions receive higher priority.

When a trace interrupt is taken, instruction execution for the handler begins at offset 0x00D00 from the base address indicated by MSR[IP].

5.5.12.1 Single-Step Instruction Trace Mode

The single-step instruction trace mode is enabled by setting MSR[SE]. Encountering the single-step breakpoint causes the following action—trap to address vector 0x00D00.

The single-step trace action traps after an instruction execution and completion.

5.5.12.2 Branch Trace Mode

The branch trace mode is enabled by setting MSR[BE]. Encountering the branch trace breakpoint causes the following action—trap to interrupt vector 0x00D00.

The branch trace action is to trap after the completion of any branch instruction whenever MSR[BE] is set.

5.5.13 Performance Monitor Interrupt (0x00F00)

The performance monitor interrupt is triggered by an enabled condition or event. The only performance monitor enabled condition or event defined for the e300c3 is the following: A PMC*n* overflow condition occurs when both of the following are true:

- The counter's overflow condition is enabled; PMLCa*n*[CE] is set.
- The counter indicates an overflow; PMC*n*[OV] is set.

If PMGC0[PMIE] is set, an enabled condition or event triggers the signaling of a performance monitor exception.

If PMGC0[FCECE] is set, an enabled condition or event also triggers all performance monitor counters to freeze.

5.5.14 Instruction TLB Miss Interrupt (0x01000)

When the effective address for an instruction load, store, or cache operation cannot be translated by the ITLB, an instruction TLB miss interrupt is generated. Register settings for the instruction and data TLB miss interrupts are described in Table 5-21.

If the instruction TLB miss interrupt fails to find the desired PTE, then a page fault must be synthesized. The handler must restore the machine state and clear MSR[TGPR] before invoking the ISI interrupt (0x00400).

Software table search operations are discussed in Chapter 6, "Memory Management."

When an instruction TLB miss interrupt is taken, instruction execution for the handler begins at offset 0x01000 from the physical base address indicated by MSR[IP].

5.5.15 Data TLB Miss on Load Interrupt (0x01100)

When the effective address for a data load or cache operation cannot be translated by the DTLB, a data TLB miss on load interrupt is generated. Register settings for the instruction and data TLB miss interrupts are described in Table 5-21.

If a data TLB miss interrupt fails to find the desired PTE, then a page fault must be synthesized. The handler must restore the machine state and clear MSR[TGPR] before invoking the DSI interrupt (0x00300).

Software table search operations are discussed inChapter 6, "Memory Management."

When a data TLB miss on load interrupt is taken, instruction execution for the handler begins at offset 0x01100 from the physical base address indicated by MSR[IP].

Table 5-21. Instruction and Data TLB Miss Interrupts—Register Settings

Register	Setting Description
SRR0	Set to the address of the next instruction to be executed in the program for which the TLB miss interrupt was generated.
SRR1	0–3 Loaded from condition register CR0 field 4–11 Cleared 12 KEY. Key for TLB miss (SR[Ks] or SR[Kp], depending on whether the access is a user or supervisor access). 13 D/I. Data or instruction access. 0 = Data TLB miss 1 = Instruction TLB miss 14 WAY. Next TLB set to be replaced (set per LRU). 0 = Replace TLB associativity set 0 1 = Replace TLB associativity set 1 15 S/L. Store or load data access. 0 = Data TLB miss on load 1 = Data TLB miss on store (or C = 0) 16–31 Loaded from MSR[16–31]
MSR	POW 0 FP 0 FE1 0 RI 0 TGPR 1 ME — CE — LE Set to value of ILE ILE — FE0 0 IP — EE 0 SE 0 IR 0 PR 0 BE 0 DR 0

5.5.16 Data TLB Miss on Store Interrupt (0x01200)

When the effective address for a data store or cache operation cannot be translated by the DTLB, a data TLB miss on store interrupt is generated. The data TLB miss on store interrupt is also taken when the changed bit (C = 0) for a DTLB entry needs to be updated for a store operation. Register settings for the instruction and data TLB miss interrupts are described in Table 5-21.

If a data TLB miss interrupt handler fails to find the desired PTE, then a page fault must be synthesized. The handler must restore the machine state and clear MSR[TGPR] before invoking the DSI interrupt (0x00300).

Software table search operations are discussed in Chapter 6, “Memory Management.”

When a data TLB miss on store interrupt is taken, instruction execution for the handler begins at offset 0x01200 from the physical base address indicated by MSR[IP].

5.5.17 Instruction Address Breakpoint Interrupt (0x01300)

The instruction address breakpoint is controlled by the IABR special purpose register. Bits [0–29] of IABR holds an effective address to which each instruction’s address is compared. The interrupt is enabled by setting bit 30 in the IABR. The interrupt is taken when an instruction breakpoint address matches on the next instruction to complete. The instruction tagged with the match is not completed before the instruction address breakpoint interrupt is taken.

The breakpoint action can be trapped to interrupt vector 0x01300 (default).

Note that the e300 core has a second instruction address breakpoint register, IABR2, that functions identically to IABR, and allows for two instruction breakpoints to be enabled.

The bit settings for when an instruction address breakpoint interrupt is taken are shown in Table 5-22.

Table 5-22. Instruction Address Breakpoint Interrupt—Register Settings

Register	Setting Description			
SRR0	Set to the address of the next instruction to be executed in the program for which the TLB miss interrupt was generated.			
SRR1	0–15 Cleared 16–31 Loaded from MSR[16–31]			
MSR	POW 0 TGPR 0 ILE — EE 0 PR 0	FP 0 ME — FE0 0 SE 0 BE 0	FE1 0 CE — IP — IR 0 DR 0	RI 0 LE Set to value of ILE

The default breakpoint action is to trap before the execution of the matching instruction.

Table 5-23 shows the priority of actions taken when more than one mode is enabled for the same instruction.

If trace and breakpoint conditions occur simultaneously, the breakpoint conditions receive higher priority.

The e300 core requires that an **mtspr** instruction that updates the IABR be followed by a context-synchronizing instruction. If the **mtspr** instruction enables the instruction address breakpoint interrupt, the context-synchronizing instruction cannot generate a breakpoint response. The e300 core also cannot block a breakpoint response on the context-synchronizing instruction if the breakpoint was disabled by the **mtspr** instruction. See “Synchronization Requirements for Special Registers and TLBs” in Chapter 2, “Register Set,” in the *Programming Environments Manual*, for more information on this requirement.

Table 5-23. Breakpoint Action for Multiple Modes Enabled for the Same Address

IABR[IE]	MSR[BE]	MSR[SE]	First Action	Next Action	Comments
1	1	0	Instruction address breakpoint	Trace (branch)	Enabling both modes is useful only if both trace and address breakpoint interrupts are needed.
1	0	1	Instruction address breakpoint	Trace (single-step)	Enabling both modes is useful only if different breakpoint actions are required.
0	1	1	Trace (branch)	None	The action for branch trace and single-step trace is the same. Enabling both trace modes is redundant except for hard stop on branches.
1	1	1	Instruction address breakpoint	Trace	Enabling all modes is redundant. This entry is for clarification only.

Section 2.2.14, “Instruction Address Breakpoint Registers (IABR and IABR2),” and Chapter 10, “Debug Features,” provide more information about the instruction breakpoint facility.

5.5.18 System Management Interrupt (0x01400)

The system management interrupt behaves like an external interrupt except for the signal asserted and the vector taken. A system management interrupt is signaled to the e300 core by the assertion of the *smi* signal. The interrupt may not be recognized if a higher priority interrupt occurs simultaneously or if MSR[EE] is cleared when *smi* is asserted. Note that *smi* takes priority over *int* if they are recognized simultaneously.

After the *smi* is detected (and provided that MSR[EE] is set), the e300 core generates a recoverable halt to instruction completion. The e300 core requires the next instruction in program order to complete or except, block completion of any following instructions, and allow the completed store queue to drain (see Section 7.1, "Terminology and Conventions," for the definition). If any higher priority interrupts are encountered in this process, they are taken first and the system management interrupt is delayed until a recoverable halt is achieved. At this time the e300 core saves state information and takes the system management interrupt.

The register settings for the external interrupt are shown in Table 5-24.

Table 5-24. System Management Interrupt—Register Settings

Register	Setting Description			
SRR0	Set to the effective address of the instruction that the processor would have attempted to complete next if no interrupt conditions were present.			
SRR1	0–15 Cleared 16–31 Loaded from MSR[16–31]			
MSR	POW 0 TGPR 0 ILE — EE 0 PR 0	FP 0 ME — FE0 0 SE 0 BE 0	FE1 0 CE — IP — IR 0 DR 0	RI 0 LE Set to value of ILE

When a system management interrupt is taken, instruction execution for the handler begins at offset 0x01400 from the physical base address indicated by MSR[IP].

The e300 core recognizes the interrupt condition (*smi* asserted) only if the MSR[EE] bit is set; otherwise, the interrupt condition is ignored. To guarantee that the external interrupt is taken, the *smi* signal must be held active until the e300 core takes the interrupt. If the *smi* signal is negated before the interrupt is taken, the e300 core is not guaranteed to take a system management interrupt. The interrupt handler must send a command to the device that asserted *smi*, acknowledging the interrupt and instructing the device to negate *smi*.

Chapter 6
Memory Management

This chapter describes the e300 core implementation of the memory management unit (MMU) specifications provided by the PowerPC operating environment architecture (OEA). The MMU implementation of the e300 core is the same as that of previous PowerPC microprocessors. However, the e300 core implements four additional IBAT entries and four additional DBAT entries.

The primary function of the MMU in a processor of this family is the translation of logical (effective) addresses to physical addresses (referred to as real addresses in the architecture specification) for memory accesses, and I/O accesses (I/O accesses are assumed to be memory-mapped). In addition, the MMU provides access protection on a segment, block, or page basis. This chapter describes the specific hardware used to implement the MMU model of the OEA in the core. Refer to Chapter 7, "Memory Management," in the *Programming Environments Manual* for a complete description of the conceptual model.

Two general types of accesses generated by processors that implement the PowerPC architecture require address translation—instruction accesses, and data accesses to memory generated by load and store instructions. Generally, the address translation mechanism is defined in terms of segment descriptors and page tables defined by the PowerPC architecture for locating the effective-to-physical address mapping for instruction and data accesses. The segment information translates the effective address to an interim virtual address and the page table information translates the virtual address to a physical address.

The segment descriptors, used to generate the interim virtual addresses, are stored as on-chip segment registers on 32-bit implementations (such as the e300 core). In addition, two translation lookaside buffers (TLBs) are implemented on the core to keep recently-used page address translations on-chip. Although the OEA describes one MMU (conceptually), the core hardware maintains separate TLBs and table search resources for instruction and data accesses that can be accessed independently (and simultaneously). Therefore, the core is described as having two MMUs, one for instruction accesses (IMMU) and one for data accesses (DMMU).

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations on-chip. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor-level special-purpose registers (SPRs). There are separate instruction and data BAT mechanisms, and in the e300 core, they reside in the instruction and data MMUs, respectively.

The MMUs, together with the interrupt processing mechanism, provide the necessary support for the operating system to implement a paged virtual memory environment and for enforcing protection of designated memory areas. Interrupt processing is described in Chapter 5, "Interrupts and Exceptions." Section 5.2, "Interrupt Processing," describes the MSR which controls some of the critical functionality of the MMUs.

6.1 MMU Features

The e300 core completely implements all features required by the memory management specification of the OEA for 32-bit implementations. Thus, it provides 4 Gbytes of effective address space accessible to supervisor and user programs with a 4-Kbyte page size and 256-Mbyte segment size. In addition, the MMUs of 32-bit processors use an interim virtual address (52 bits) and hashed page tables in the generation of 32-bit physical addresses. These processors also have a BAT mechanism for mapping large blocks of memory. Block sizes range from 128 Kbytes to 256 Mbytes and are software-programmable.

Table 6-1 summarizes all e300 core MMU features including the architectural features of PowerPC MMUs (defined by the OEA) for 32-bit processors and the implementation-specific features provided by the core.

Table 6-1. MMU Features Summary

Feature Category	Architecturally Defined/ e300 Core-Specific	Feature
Address ranges	Architecturally defined	2^{32} bytes of effective address
		2^{52} bytes of virtual address
		2^{32} bytes of physical address
Page size	Architecturally defined	4 Kbytes
Segment size	Architecturally defined	256 Mbytes
Block address translation	Architecturally defined	Range of 128 Kbytes–256 Mbytes sizes
		Implemented with IBAT and DBAT registers in BAT array
Memory protection	Architecturally defined	Segments selectable as no-execute
		Pages selectable as user/supervisor and read-only
		Blocks selectable as user/supervisor and read-only
Page history	Architecturally defined	Reference and change bits defined and maintained
Page address translation	Architecturally defined	Translations stored as PTEs in hashed page tables in memory
		Page table size determined by mask in SDR1 register
TLBs	Architecturally defined	Instructions for maintaining optional TLBs (**tlbie** instruction in G2 core)
	e300 core-specific	64-entry (32-entry byway), two-way set-associative ITLB 64-entry(32-entry byway), two-way set-associative DTLB
Segment descriptors	Architecturally defined	Stored as segment registers on-chip

Table 6-1. MMU Features Summary (continued)

Feature Category	Architecturally Defined/ e300 Core-Specific	Feature
Page table search support	e300 core-specific	Three MMU interrupts defined: ITLB miss, DTLB miss on load, and DTLB miss on store (or C = 0); MMU-related bits set in SRR1 for these interrupts.
		IMISS and DMISS registers (missed effective address) HASH1 and HASH2 registers (PTEG addr) ICMP and DCMP registers (for comparing PTEs) RPA register (for loading TLBs)
		tlbli rB instruction for loading ITLB entries **tlbld r**B instruction for loading DTLB entries
		Shadow registers for GPR0–GPR3 (can use **r0**–**r3** in table search handler without corruption of **r0**–**r3** in context that was previously executing), called TGPR0–TGPR3.

6.1.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, or cache instruction, and when it fetches the next instruction. The effective address is translated to a physical address according to the procedures described in Chapter 7, “Memory Management,” in the *Programming Environments Manual*, augmented with information in this chapter. The memory subsystem uses the physical address for the access.

For a complete discussion of effective address calculation, see Section 3.2.2.3, “Effective Address Calculation.”

6.1.2 MMU Organization

Figure 6-1 shows the conceptual organization of a PowerPC MMU in a 32-bit implementation; note that it does not describe the specific hardware used to implement the memory management function for a particular processor. Processors may optionally implement on-chip TLBs and may optionally support the automatic search of the page tables for PTEs. In addition, other hardware features (invisible to the system software) not depicted in the figure may be implemented.

Figure 6-2 and Figure 6-3 show the conceptual organization of the core instruction and data MMUs, respectively. The instruction addresses shown in Figure 6-2 are generated by the processor for sequential instruction fetches and addresses that correspond to a change of program flow. Data addresses shown in Figure 6-3 are generated by load and store instructions and by cache instructions.

As shown in the figures, after an address is generated, the higher-order bits of the effective address, EA0–EA19 (or a smaller set of address bits, EA0–EA*n*, in the cases of blocks), are translated into physical address bits PA0–PA19. The lower-order address bits, A20–A31, are untranslated and, therefore, identical for both effective and physical addresses. After translating the address, the MMUs pass the resulting 32-bit physical address to the memory subsystem.

In addition to the higher-order address bits, the MMUs automatically keep an indicator of whether each access was generated as an instruction or data access and a supervisor/user indicator that reflects the state of the PR bit of the MSR when the effective address was generated. In addition, for data accesses, there is an indicator of whether the access is for a load or a store operation. This information is then used by the MMUs to appropriately direct the address translation and to enforce the protection hierarchy programmed by the operating system. Section 5.2, “Interrupt Processing,” describes the MSR, which controls some of the critical functionality of the MMUs.

The figures show how the A20–A26 address bits index into the on-chip instruction and data caches to select a cache set. The remaining physical address bits are then compared with the tag fields (comprised of bits PA0–PA19) of the four selected cache blocks to determine if a cache hit has occurred. In the case of a cache miss, the instruction or data access is then forwarded to the bus interface unit which then initiates a Coherent System Bus (CSB) access to the memory subsystem.

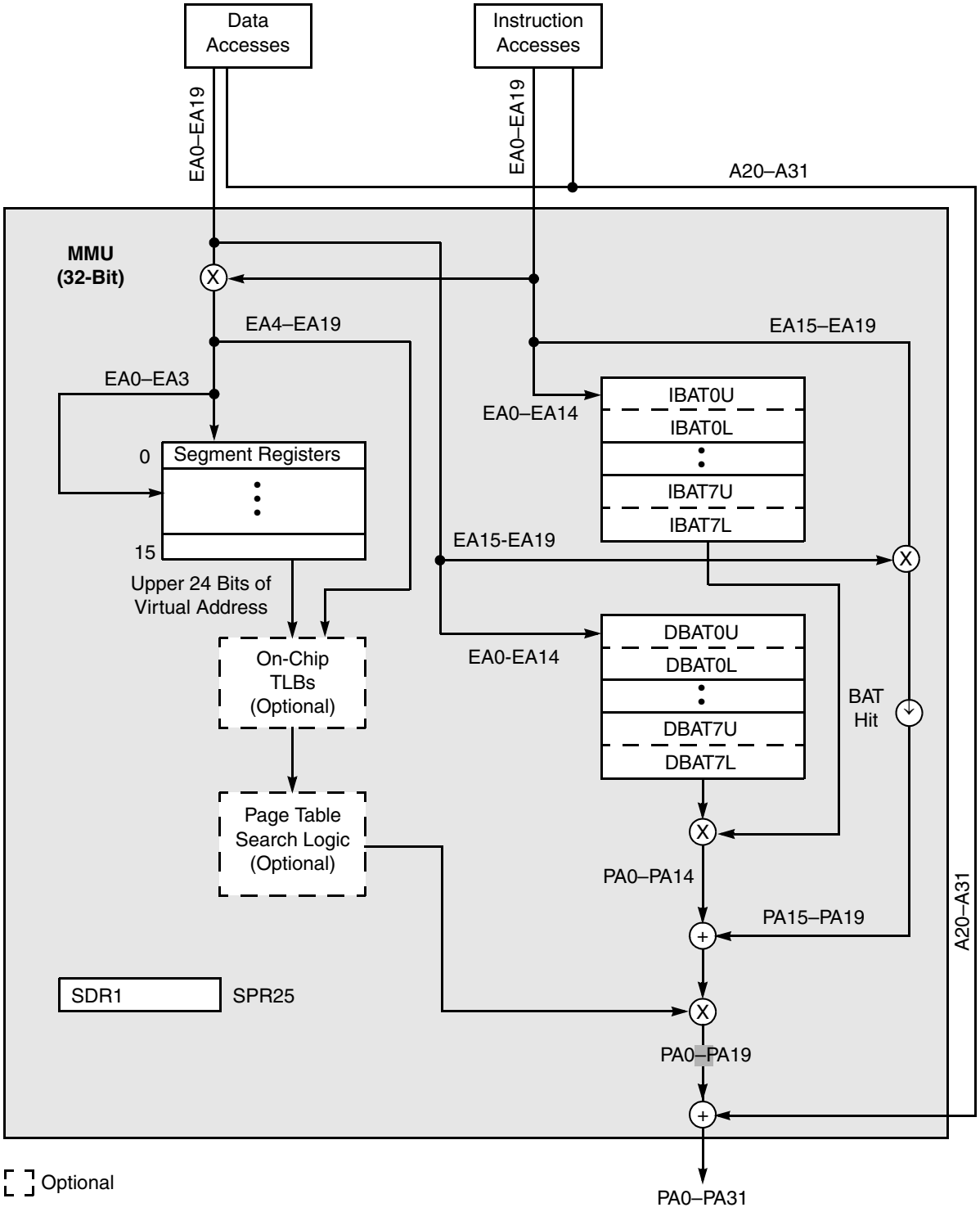


Figure 6-1. MMU Conceptual Block Diagram—32-Bit Implementations

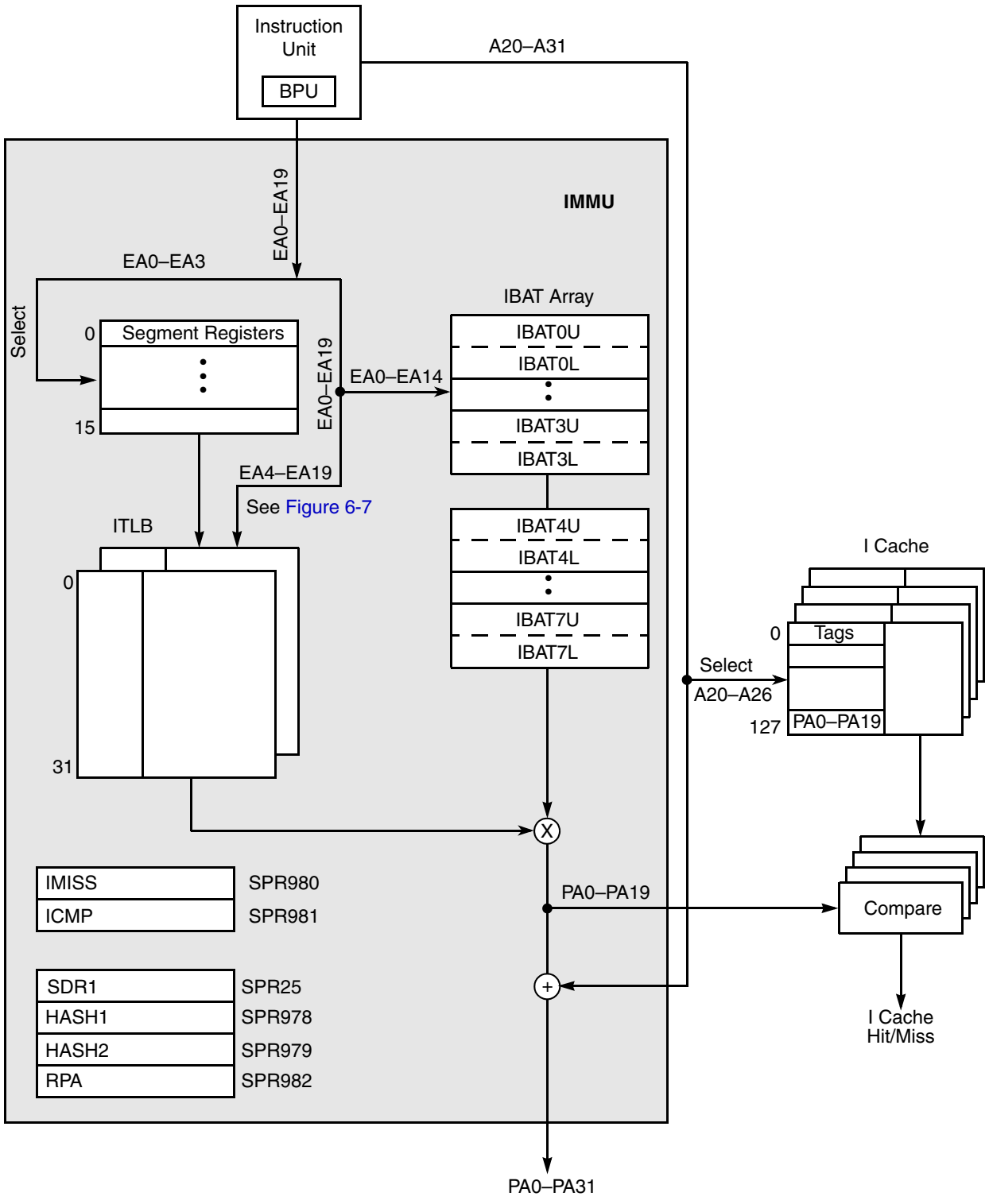


Figure 6-2. e300 Core IMMU Block Diagram

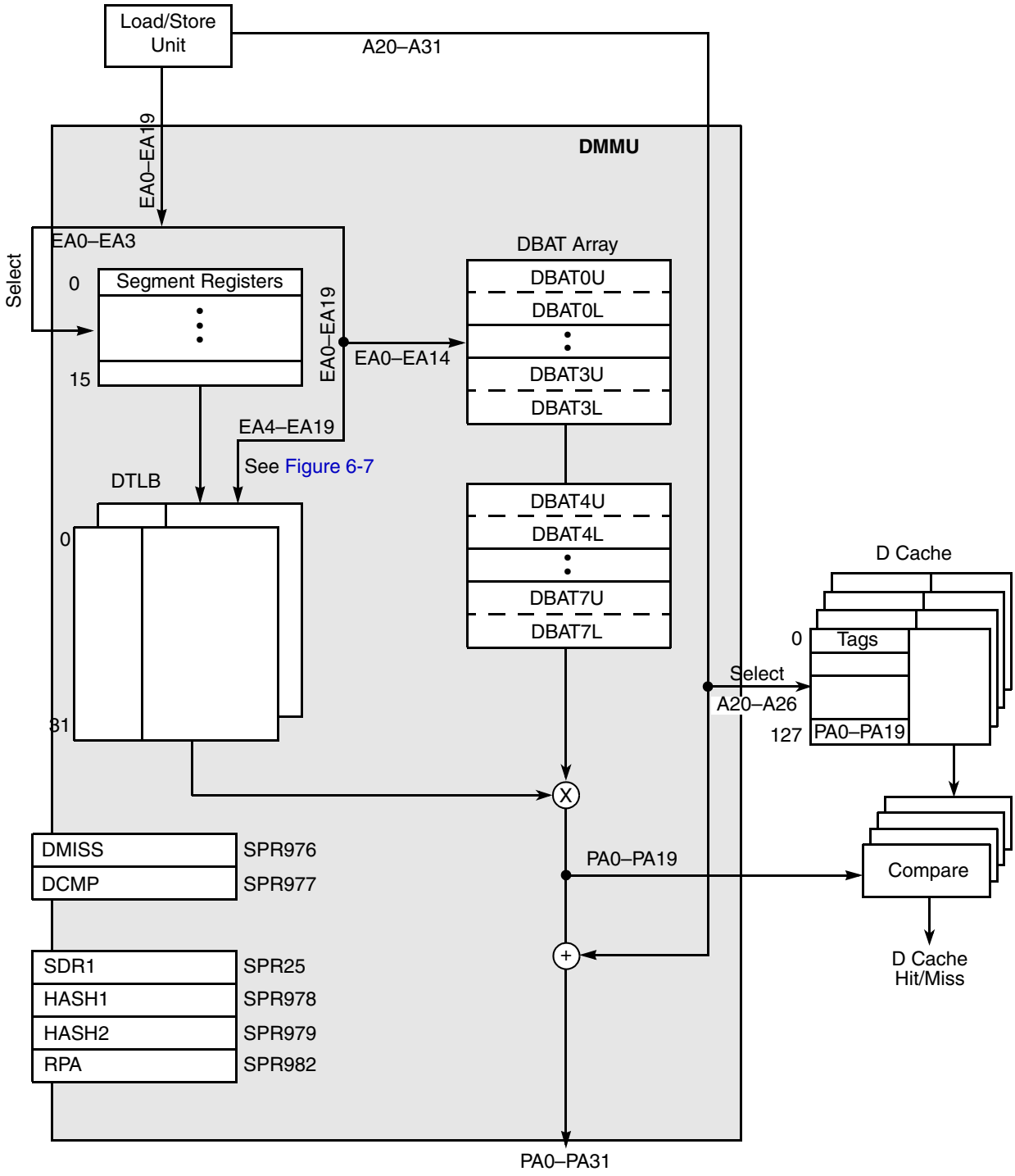


Figure 6-3. e300 Core DMMU Block Diagram

6.1.3 Address Translation Mechanisms

Processors that implement the PowerPC architecture support the following four types of address translation:

- Page address translation—translates the page frame address for a 4-Kbyte page size.
- Block address translation—translates the block number for blocks that range in size from 128 Kbytes to 256 Mbytes.
- Direct-store interface address translation—used to generate direct-store interface accesses on the external bus; not implemented in the e300 core.
- Real addressing mode translation—when address translation is disabled, the physical address is identical to the effective address.

Figure 6-4 shows the three implemented address translation mechanisms provided by the MMUs. The segment descriptors shown in the figure, control the page address translation mechanism. When an access uses page address translation, the appropriate segment descriptor is required. In 32-bit implementations, one of the 16 on-chip segment registers (which contain segment descriptors) is selected by the 4 highest-order effective address bits.

A control bit in the corresponding segment descriptor then determines if the access is to memory (memory-mapped) or to the direct-store interface space (selected when the direct-store translation control bit (T bit) in the corresponding segment descriptor is set). Note that the direct-store interface existed in previous processors only for compatibility with I/O devices that use this interface. When an access is determined to be to the direct-store interface space, the core takes a DSI interrupt as described in Section 5.5.3, “DSI Interrupt (0x00300),” if it is a data access. The G2 core takes an ISI interrupt as described in Section 5.5.4, “ISI Interrupt (0x00400),” if it is an instruction access.

For memory accesses translated by a segment descriptor, the interim virtual address is generated using the information in the segment descriptor. Page address translation corresponds to the conversion of this virtual address into the 32-bit physical address used by the memory subsystem. In most cases, the physical address for the page resides in an on-chip TLB and is available for quick access. However, if the page address translation misses in an on-chip TLB, the MMU causes a search of the page tables in memory (using the virtual address information and a hashing function) to locate the required physical address. When this occurs, the core vectors to the interrupt handlers that search the page tables with software.

Block address translation occurs in parallel with page address translation and is similar to page address translation; however, fewer higher-order effective address bits are translated into physical address bits (more lower-order address bits (at least 17) are untranslated to form the offset into a block). Also, instead of segment descriptors and a TLB, block address translations use the on-chip BAT registers as a BAT array. If an effective address matches the corresponding field of a BAT register, the information in the BAT register is used to generate the physical address; in this case, the results of the page translation (occurring in parallel) are ignored (even if the segment corresponds to the direct-store interface space).

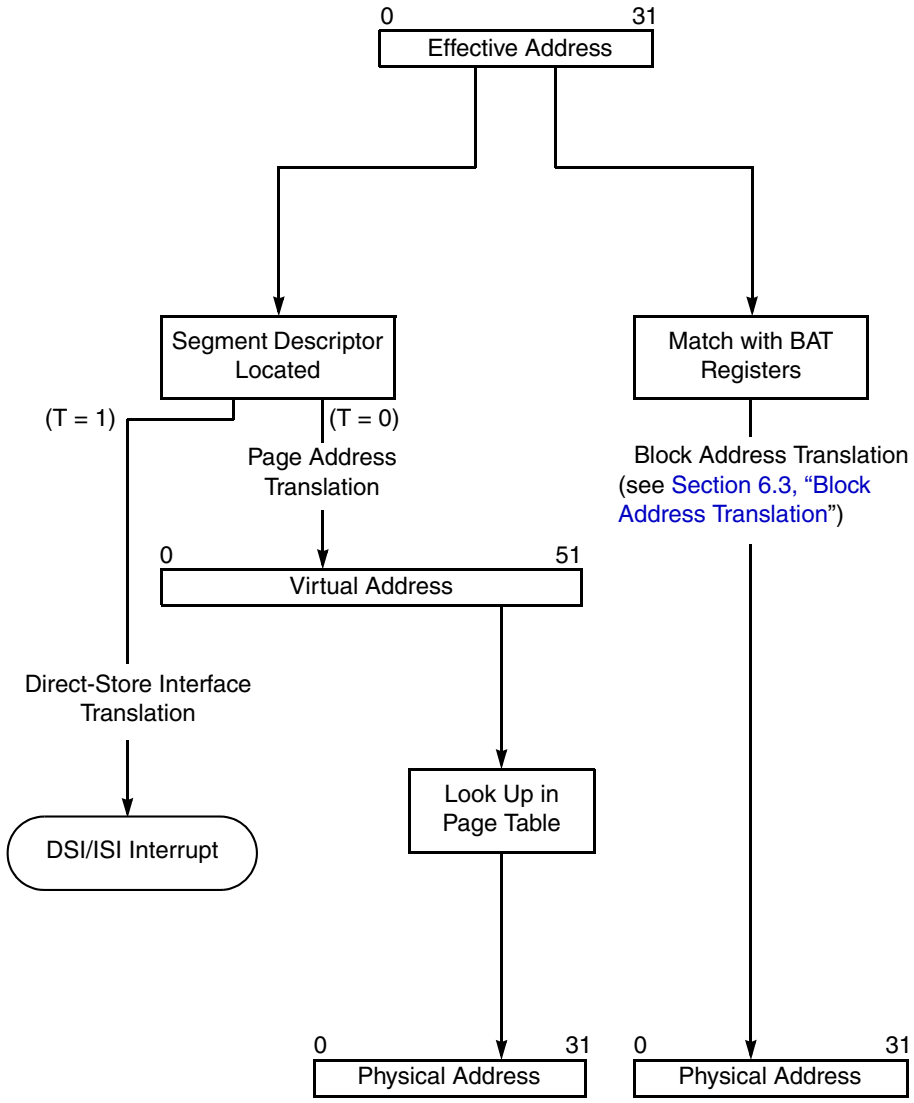


Figure 6-4. Address Translation Types

6.1.4 Memory Protection Facilities

In addition to the translation of effective addresses to physical addresses, the MMUs provide access protection of supervisor areas from user access and can designate areas of memory as read-only, as well as, no-execute or guarded. Table 6-2 shows the eight protection options supported by the MMUs for pages.

Table 6-2. Access Protection Options for Pages

Option	User Read		User Write	Supervisor Read		Supervisor Write
	I-Fetch	Data		I-Fetch	Data	
Supervisor-only	—	—	—	√	√	√
Supervisor-only-no-execute	—	—	—	—	√	√

Table 6-2. Access Protection Options for Pages (continued)

Option	User Read		User Write	Supervisor Read		Supervisor Write
	I-Fetch	Data		I-Fetch	Data	
Supervisor-write-only	√	√	—	√	√	√
Supervisor-write-only-no-execute	—	√	—	—	√	√
Both user/supervisor	√	√	√	√	√	√
Both user/supervisor-no-execute	—	√	√	—	√	√
Both read-only	√	√	—	√	√	—
Both read-only-no-execute	—	√	—	—	√	—

Note:
√ access permitted.
— protection violation.

The operating system programs whether instructions can be fetched from an area of memory by appropriately using the no-execute option provided in the segment descriptor. Each of the remaining options is enforced, based on a combination of information in the segment descriptor and the page table entry. Thus, the supervisor-only option allows only read and write operations generated while the processor is operating in supervisor mode (corresponding to MSR[PR] = 0) to access the page. User accesses that map into a supervisor-only page cause an interrupt to be taken.

Finally, there is a facility in the VEA and OEA that allows pages or blocks to be designated as guarded, preventing out-of order accesses that may cause undesired side effects. For example, areas of the memory map that are used to control I/O devices can be marked as guarded so that accesses (for example, instruction prefetches) do not occur unless they are explicitly required by the program.

For more information on memory protection, see “Memory Protection Facilities” in Chapter 7, “Memory Management,” in the *Programming Environments Manual*.

6.1.5 Page History Information

The MMUs of these processors also define reference (R) and change (C) bits in the page address translation mechanism that can be used as history information relevant to the page. This information can then be used by the operating system to determine the areas of memory to write back to disk when new pages must be allocated in main memory. While these bits are initially programmed by the operating system into the page table, the architecture specifies that the R and C bits may be maintained either by the processor hardware (automatically) or by some software-assist mechanism that updates these bits when required as needed by the core. The software table search routines used by the core set the R bit when a PTE is accessed; the core causes an interrupt (to vector to the software table search routines) when the C bit in the corresponding TLB entry requires updating. See Section 6.4.1.3, “Scenarios for Reference and Change Bit Recording,” for more details.

6.1.6 General Flow of MMU Address Translation

The following sections describe the general flow used by processors that implement the PowerPC architecture to translate effective addresses to virtual and then physical addresses.

6.1.6.1 Real Addressing Mode and Block Address Translation Selection

When an instruction or data access is generated and the corresponding instruction or data translation is disabled (MSR[IR] = 0 or MSR[DR] = 0), real addressing mode translation is used (physical address equals effective address) and the access continues to the memory subsystem as described in Section 6.2, "Real Addressing Mode."

Figure 6-5 shows the flow used by the MMUs in determining whether to select real addressing mode, block address translation, or to use the segment descriptor to select page address translation.

Note that if the BAT array search results in a hit, the access is qualified with the appropriate protection bits. If the access violates the protection mechanism, an interrupt (ISI or DSI interrupt) is generated.

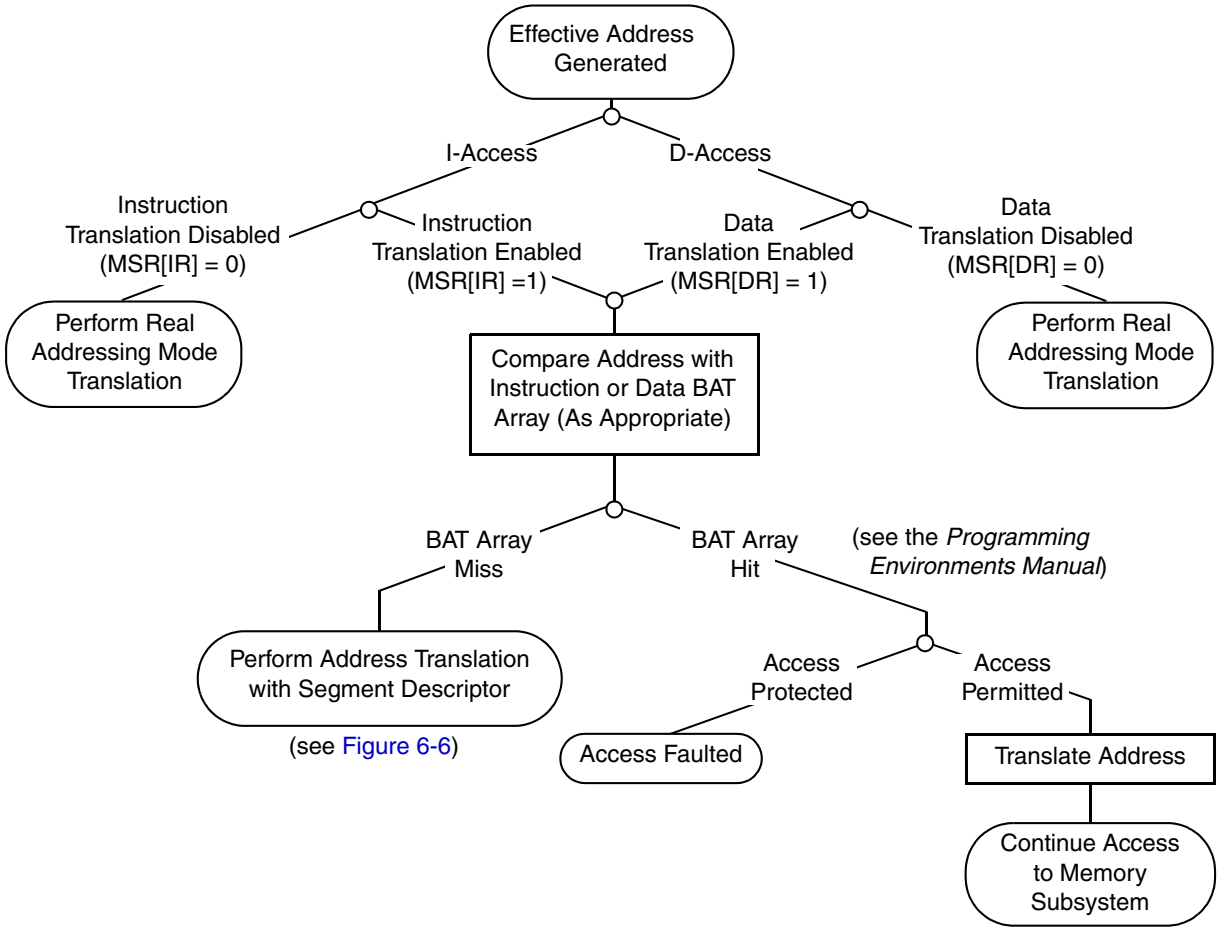

Effective Address Generated
I-Access
D-Access
Instruction Translation Disabled (MSR[IR] = 0)
Instruction Translation Enabled (MSR[IR] =1)
Data Translation Enabled (MSR[DR] = 1)
Data Translation Disabled (MSR[DR] = 0)
Perform Real Addressing Mode Translation
Compare Address with Instruction or Data BAT Array (As Appropriate)
Perform Real Addressing Mode Translation
BAT Array Miss
BAT Array Hit
(see the Programming Environments Manual)
Perform Address Translation with Segment Descriptor
(see Figure 6-6)
Access Protected
Access Permitted
Access Faulted
Translate Address
Continue Access to Memory Subsystem

Figure 6-5. General Flow of Address Translation (Real Addressing Mode and Block)

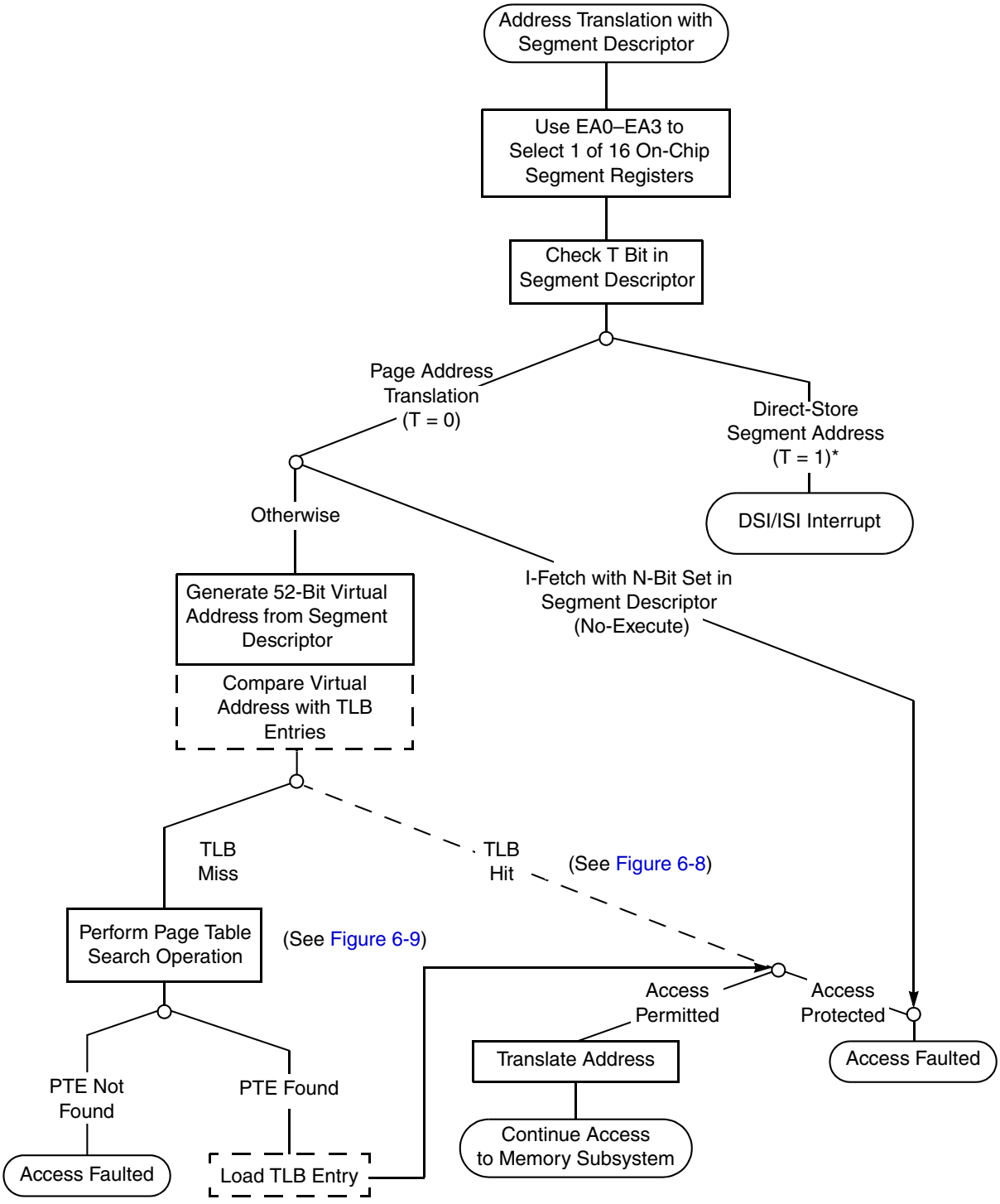
6.1.6.2 Page Address Translation Selection

If address translation is enabled (real addressing mode not selected) and the effective address information does not match with a BAT array entry, then the segment descriptor must be located. Once the segment descriptor is located, the T bit in the segment descriptor selects whether the translation is to a page or to a direct-store interface segment, as shown in Figure 6-6. Note that the e300 core does not implement the direct-store interface, and accesses to these segments cause a DSI interrupt. In addition, Figure 6-6 also shows the way the no-execute protection is enforced; if the N bit in the segment descriptor is set and the access is an instruction fetch, the access is faulted as described in Chapter 7, “Memory Management,” in the *Programming Environments Manual*. Note that the figure shows the flow for these cases as described by the OEA and, therefore, the TLB references are shown as optional. Since the core implements TLBs, these branches are valid, and described in more detail throughout this chapter.

If the T bit in the corresponding segment descriptor is zero, page address translation is selected. The information in the segment descriptor is then used to generate the 52-bit virtual address. The virtual address is then used to identify the page address translation information (stored as page table entries (PTEs) in a page table in memory). For increased performance, the core has two TLBs to store recently-used PTEs on-chip.

If an access hits in the appropriate TLB, the page translation occurs and the physical address bits are forwarded to the memory subsystem. If the required PTE is not resident, the MMU requires a search of the page table. In this case, the core traps to one of three interrupt handlers for the system software to perform the page table search. If the PTE is successfully matched, a new TLB entry is created and the page translation is once again attempted. This time, the TLB is guaranteed to hit. Once the PTE is located, the access is qualified with the appropriate protection bits. If the access is a protection violation (not allowed), an interrupt (instruction access or data access) is generated.

If the PTE is not found by the table search operation, a page fault condition exists, and the TLB miss interrupt handlers synthesize either an ISI or DSI interrupt to handle the page fault.


Address Translation with Segment Descriptor
Use EA0–EA3 to Select 1 of 16 On-Chip Segment Registers
Check T Bit in Segment Descriptor
Page Address Translation (T = 0)
Direct-Store Segment Address (T = 1)*
DSI/ISI Interrupt
Otherwise
I-Fetch with N-Bit Set in Segment Descriptor (No-Execute)
Generate 52-Bit Virtual Address from Segment Descriptor
Compare Virtual Address with TLB Entries
TLB Miss
TLB Hit
(See Figure 6-8)
Perform Page Table Search Operation
(See Figure 6-9)
Access Permitted
Access Protected
Translate Address
Access Faulted
PTE Not Found
PTE Found
Continue Access to Memory Subsystem
Access Faulted
Load TLB Entry

– – – Optional to the PowerPC architecture. Implemented in the e300

*In the case of instruction accesses, causes ISI interrupt.

Figure 6-6. General Flow of Page and Direct-Store Interface Address Translation

6.1.7 MMU Interrupts Summary

In order to complete any memory access, the effective address must be translated to a physical address. In the e300 core, an MMU interrupt condition occurs if this translation fails for one of the following reasons:

- Page fault—There is no valid page table entry to identify the page specified by the effective address (and segment descriptor) and there is no valid BAT translation.
- An address translation is found but the access is not allowed by the memory protection mechanism.

Additionally, because the core relies on software to perform table search operations, the processor also takes an interrupt when one of the following occurs:

- There is a miss in the corresponding (instruction or data) TLB.
- The page table requires an update to the change (C) bit.

The state saved by the processor for each of these interrupts contains information that identifies the address of the failing instruction. Refer to Chapter 5, “Interrupts and Exceptions,” for a more detailed description of interrupt processing.

Because a page fault condition (PTE not found in the page tables in memory) is detected by the software that performs the table search operation (and not the core hardware), it does not cause a core interrupt, in the strictest sense, in that interrupt processing as described in Chapter 5, “Interrupts and Exceptions,” does not occur. However, in order to maintain architectural compatibility with software written for other devices that implement the PowerPC architecture, the software that detects this condition should synthesize an interrupt by setting the appropriate bits in the DSISR or SRR1 and branching to the ISI or DSI handler. Refer to Section 6.5.2, “Implementation-Specific Table Search Operation,” for more information and examples of this interrupt software. The remainder of this chapter assumes that the table search software emulates this interrupt and refers to this condition as an interrupt.

The translation exception conditions defined by the OEA for 32-bit implementations cause either the ISI or the DSI interrupt to be taken as shown in Table 6-3.

Table 6-3. Translation Exception Conditions

Exception Condition	Description	Interrupt
Page fault (no PTE found)	No matching PTE found in page tables (and no matching BAT array entry)	I access: ISI interrupt[1] SRR1[1] = 1
		D access: DSI interrupt [1] DSISR[1] =1
Block protection violation	Conditions described for block in “Block Memory Protection” in Chapter 7, “Memory Management,” in the *Programming Environments Manual*.“	I access: ISI interrupt SRR1[4] = 1
		D access: DSI interrupt DSISR[4] =1
Page protection violation	Conditions described for page in “Page Memory Protection” in Chapter 7, “Memory Management,” in the *Programming Environments Manual.*	I access: ISI interrupt[2] SRR1[4] = 1
		D access: DSI interrupt [2] DSISR[4] =1

Table 6-3. Translation Exception Conditions (continued)

Exception Condition	Description	Interrupt
No-execute protection violation	Attempt to fetch instruction when SR[N] = 1	ISI interrupt SRR1[3] = 1
Instruction fetch from direct-store segment	Attempt to fetch instruction when SR[T] = 1	ISI interrupt SRR1[3] =1
Data access to direct-store segment (including floating-point accesses[3])	Attempt to perform load or store (including floating-point load or store) when SR[T] = 1	DSI interrupt DSISR[5] =1
Instruction fetch from guarded memory with MSR[IR] = 1	Attempt to fetch instruction when MSR[IR] = 1 and either matching xBAT[G] = 1, or no matching BAT entry and PTE[G] = 1.	ISI interrupt SRR1[3] =1

1 The e300 core hardware does not vector to these interrupts automatically. It is assumed that the software that performs the table search operation vectors to these interrupts and sets the appropriate bits when a page fault condition occurs.

2 The table search software can also vector to these exception conditions.

3 Floating-point not supported on the e300c2.

In addition to the translation exceptions, there are other MMU-related conditions (some of them defined as implementation-specific and, therefore, not required by the architecture) that can cause an interrupt to occur in the core. These exception conditions map to the processor interrupt as shown in Table 6-4. For example, the core also defines three exception conditions to support software table searching. The only exception conditions that occur for data accesses when MSR[DR] = 0, are the conditions that cause the alignment interrupt. For more detailed information about the conditions that cause the alignment interrupt (in particular for string/multiple instructions), see Section 5.5.6, “Alignment Interrupt (0x00600).”

Table 6-4. Other MMU Exception Conditions

Exception Condition	Description	Interrupt
TLB miss for an instruction fetch	No matching entry found in ITLB	Instruction TLB miss interrupt SRR1[13] = 1 MSR[14] = 1
TLB miss for a data load access	No matching entry found in DTLB for data load access	Data TLB miss on load interrupt SRR1[13] = 0 SRR1[15] = 1 MSR[14] = 1
TLB miss for a data store, or store and C = 0	No matching entry found in DTLB for data store access or matching DLTB entry has C = 0 and the access is a store	Data TLB miss on store interrupt, or store and C = 0 SRR1[13] = 0 SRR1[15] =0 MSR[14] = 1
dcbz with W = 1 or I = 1	**dcbz** instruction to write-through or cache-inhibited segment or block	Alignment interrupt (not required by architecture for this condition)
dcbz when the data cache is locked	The **dcbz** instruction takes an alignment interrupt if the data cache is locked (HID0 bits 18 and 19) when it is executed	Alignment interrupt
lwarx or stwcx. instruction to direct-store segment	Reservation instruction or external control instruction when SR[T] =1	DSI interrupt DSISR[5] = 1

Table 6-4. Other MMU Exception Conditions (continued)

Exception Condition	Description	Interrupt
Floating-point load or store to direct-store segment[1]	FP memory access when SR[T] = 1	See data access to direct-store segment in Table 6-3
Load or store that results in a direct-store error	Does not occur in G2 core	Does not apply
lmw, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** instruction attempted in little-endian mode	**lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** instruction attempted while MSR[LE] = 1.	Alignment interrupt
Operand misalignment	Translation enabled and operand is misaligned as described in Chapter 5, "Interrupts and Exceptions."	Alignment interrupt (some of these cases are implementation-specific)

[1] Floating-point not supported on the e300c2.

Note that some exception conditions depend on whether the memory area is set up as write-through (W = 1) or cache-inhibited (I = 1). These bits are described fully in "Memory/ Cache Access Attributes" in Chapter 5, "Cache Model and Memory Coherency," in the *Programming Environments Manual*. Refer to Chapter 5, "Interrupts and Exceptions," and to Chapter 6, "Interrupts," in the *Programming Environments Manual* for a complete description of the SRR1 and DSISR settings.

6.1.8 MMU Instructions and Register Summary

The MMU instructions and registers provide the operating system with the ability to set up the block address translation areas and the page tables in memory.

Note that because the implementation of TLBs is optional, the instructions that refer to these structures are also optional. However, because these structures serve as caches of the page table, the architecture specifies a software protocol for maintaining coherency between these caches and the tables in memory whenever changes are made to the tables in memory. When the tables in memory are changed, the operating system purges these caches of the corresponding entries, allowing the translation caching mechanism to refetch from the tables when the corresponding entries are required.

Note that the e300 core implements all TLB-related instructions except **tlbia**, which is treated as an illegal instruction. The core also uses some implementation-specific instructions to load two on-chip TLBs.

Because the MMU specification for these processors is so flexible, it is recommended that the software that uses these instructions and registers be encapsulated into subroutines to minimize the impact of migrating across the family of implementations.

Table 6-5 summarizes e300 core instructions that specifically control the MMU. For more detailed information about the instructions, refer to Chapter 3, "Instruction Set Model," in this book and Chapter 8, "Instruction Set," in the *Programming Environments Manual*.

Table 6-5. Instruction Summary—MMU Control

Instruction	Description
mtsr SR,**r**S	Move to Segment Register SR[SR#]← **r**S
mtsrin rS,**r**B	Move to Segment Register Indirect SR[**r**B[0–3]]← **r**S
mfsr rD,SR	Move from Segment Register **r**D←SR[SR#]
mfsrin rD,**r**B	Move from Segment Register Indirect **r**D←SR[**r**B[0–3]]
tlbie rB[1]	TLB Invalidate Entry For effective address specified by **r**B, TLB[V]← 0 The **tlbie** instruction invalidates both TLB entries indexed by the EA, and operates on both the instruction and data TLBs simultaneously invalidating four TLB entries. The index corresponds to bits 15–19 of the EA. Software must ensure that instruction fetches or memory references to the virtual pages specified by the **tlbie** instruction have been completed prior to executing the **tlbie** instruction.
tlbsync [1]	TLB Synchronize Synchronizes the execution of all other **tlbie** instructions in the system. In the e300 core, when the $\overline{tlbisync}$ signal is negated, instruction execution may continue or resume after the completion of a **tlbsync** instruction. When the $\overline{tlbisync}$ signal is asserted, instruction execution stops after the completion of a **tlbsync** instruction.
tlbli (implementation -specific)	Load Instruction TLB Entry Loads the contents of the ICMP and RPA registers into the ITLB.
tlbld (implementation -specific)	Load Data TLB Entry Loads the contents of the DCMP and RPA registers into the DTLB.

[1] These instructions are defined by the PowerPC architecture, but are optional.

Table 6-6 summarizes the registers that the operating system uses to program the e300 core MMUs. These registers are accessible to supervisor-level software only. These registers are described in Chapter 2, “Register Set,” in the *Programming Environments Manual.* For e300 core-specific registers, see Chapter 2, “Register Model,” of this book.

Table 6-6. MMU Registers

Register	Description
Segment registers (SR0–SR15)	The sixteen 32-bit segment registers are present only in 32-bit implementations of the PowerPC architecture. The fields in the segment register are interpreted differently depending on the value of bit 0. The segment registers are accessed by the **mtsr**, **mtsrin**, **mfsr**, and **mfsrin** instructions.
BAT registers e300: (IBAT0U–IBAT7U, IBAT0L–IBAT7L, DBAT0U–DBAT7U, and DBAT0L–DBAT7L)	The e300 core has 32 BAT registers, organized as 8 pairs of instruction BAT registers (IBAT0U–IBAT7U paired with IBAT0L–IBAT7L) and 8 pairs of data BAT registers (DBAT0U–DBAT7U paired with DBAT0L–DBAT7L). The BAT registers are defined as 32-bit registers in 32-bit implementations. These are special-purpose registers that are accessed by the **mtspr** and **mfspr** instructions, regardless of the setting of HID2[13].

Table 6-6. MMU Registers (continued)

Register	Description
SDR1	The SDR1 register specifies the variable used in accessing the page tables in memory. SDR1 is defined as a 32-bit register for 32-bit implementations. This is a special-purpose register that is accessed by the **mtspr** and **mfspr** instructions.
Instruction TLB miss address and data TLB miss address registers (IMISS and DMISS)	When a TLB miss interrupt occurs, the IMISS or DMISS register contains the 32-bit effective address of the instruction or data access, respectively, that caused the miss. Note that the e300 core always loads a big-endian address into the DMISS register. These registers are implementation-specific.
Primary and secondary hash address registers (HASH1 and HASH2)	The HASH1 and HASH2 registers contain the primary and secondary PTEG addresses that correspond to the address causing a TLB miss. These PTEG addresses are automatically derived by the core by performing the primary and secondary hashing function on the contents of IMISS or DMISS, for an ITLB or DTLB miss interrupt, respectively. These registers are implementation-specific.
Instruction and data PTE compare registers (ICMP and DCMP)	The ICMP and DCMP registers contain the word to be compared with the first word of a PTE in the table search software routine to determine if a PTE contains the address translation for the instruction or data access. The contents of ICMP and DCMP are automatically derived by the core when a TLB miss interrupt occurs. These registers are implementation-specific.
Required physical address register (RPA)	The system software loads a TLB entry by loading the second word of the matching PTE entry into the RPA register and then executing the **tlbli** or **tlbld** instruction (for loading the ITLB or DTLB, respectively). This register is implementation-specific.

Note that the core contains other features that do not specifically control the MMU, but are implemented to increase performance and flexibility. These are:

- Complete set of shadow segment registers for the instruction MMU. These registers are invisible to the programming model, as described in Section 6.4.3, “TLB Description.”
- Temporary GPR0–GPR3. These registers are available as **r0**–**r3** when MSR[TGPR] is set. The core automatically sets MSR[TGPR] whenever one of the three TLB miss interrupts occurs, allowing these interrupt handlers to have four registers that are used as scratchpad space, without having to save or restore this part of the machine state that existed when the interrupt occurred. Note that MSR[TGPR] is restored to the value in SRR1 when the **rfi** instruction is executed. Refer to Section 6.5.2, “Implementation-Specific Table Search Operation,” for code examples that take advantage of these registers.

In addition, the core also automatically saves the values of CR[CR0] of the executing context to SRR1[0–3] whenever one of the three TLB miss interrupts occurs. Thus, the interrupt handler can set CR[CR0] bits and branch accordingly in the interrupt handler routine, without having to save the existing CR[CR0] bits. However, the interrupt handler must restore these bits to CR[CR0] before executing the **rfi** instruction. There are also four other bits saved in SRR1 whenever a TLB miss interrupt occurs that give information about whether the access was an instruction or data access; and if it was a data access, whether it was for a load or a store instruction. Also, these bits give some information related to the protection attributes for the access, and which set in the TLB will be replaced when the next TLB entry is loaded. Refer to Section 6.5.2.1, “Resources for Table Search Operations,” for more information on these bits and their use.

6.2 Real Addressing Mode

If address translation is disabled (MSR[IR] = 0 or MSR[DR] = 0) for a particular access, the effective address is treated as the physical address and is passed directly to the memory subsystem as described in Chapter 7, “Memory Management,” in the *Programming Environments Manual*.

Note that the default WIMG bits (0b0011) cause data accesses to be considered cacheable (I = 0) and, thus, load and store accesses are weakly ordered. This is the case, even if the data cache is disabled in the HID0 register (as it is out of hard reset). If I/O devices require load and store accesses to occur in strict program order (strongly ordered), translation must be enabled so that the corresponding I bit can be set. Also, for instruction accesses, the default memory access mode bits (WIMG) are 0b0001. That is, instruction accesses are considered cacheable (I = 0), and the memory is guarded. Again, instruction cache accesses are considered cacheable even if the instruction cache is disabled in the HID0 register (as it is out of hard reset). The W and M bits have no effect on the instruction cache.

For information on the synchronization requirements for changes to MSR[IR] and MSR[DR], refer to “Synchronization Requirements for Special Registers and for Lookaside Buffers” in Chapter 2, “Register Set,” in the *Programming Environments Manual.*

6.3 Block Address Translation

The block address translation (BAT) mechanism in the OEA provides a way to map ranges of effective addresses larger than a single page into contiguous areas of physical memory. Such areas can be used for data that is not subject to normal virtual memory handling (paging), such as a memory-mapped display buffer or an extremely large array of numerical data.

The software model for block address translation in the e300 core is described in Chapter 7, “Memory Management,” in the *Programming Environments Manual* for 32-bit implementations. However, note that for improved performance, the e300 core contains twice as many BAT registers as previous PowerPC cores, as shown in Figure 6-2 and Figure 6-3.

Implementation Note—The BAT registers are not initialized by the hardware after the power-up or reset sequence. Consequently, all valid bits in both instruction and data BAT areas must be explicitly cleared before setting any BAT area for the first time and before enabling translation. Also, note that software must avoid overlapping blocks while updating a BAT area or areas. Even if translation is disabled, multiple BAT area hits (with the valid bits set) can corrupt the remaining portion (any bits except the valid bits) of the BAT registers.

Thus, multiple BAT hits (with valid bits set) are considered a programming error whether translation is enabled or disabled, and can lead to unpredictable results if translation is enabled, (or if translation is disabled, when translation is eventually enabled). For the case of unused BATs (if translation is to be enabled), it is sufficient precaution to simply clear the valid bits of the unused BAT entries.

6.4 Memory Segment Model

The core adheres to the memory segment model as defined in Chapter 7, “Memory Management,” in the *Programming Environments Manual* for 32-bit implementations. Memory in the OEA is divided into 256-Mbyte segments. This segmented memory model provides a way to map 4-Kbyte pages of effective

addresses to 4-Kbyte pages in physical memory (page address translation), while providing the programming flexibility afforded by a large virtual address space (52 bits).

The segment/page address translation mechanism may be superseded by the BAT mechanism described in Section 6.3, "Block Address Translation." If not, the translation proceeds in the following two steps:

1. From effective address to the virtual address (which never exists as a specific entity, but can be considered to be the concatenation of the virtual page number and the byte offset within a page).
2. From virtual address to physical address.

The following section highlights those areas of the memory segment model defined by the OEA that are specific to the e300 core.

6.4.1 Page History Recording

Reference (R) and change (C) bits reside in each PTE to keep history information about the page. They are maintained by a combination of the core hardware and the table search software. The operating system uses this information to determine which areas of memory to write back to disk when new pages must be allocated in main memory. Reference and change recording is performed only for accesses made with page address translation and not for translations made with the BAT mechanism or for accesses that correspond to direct-store interface (T = 1) segments. Furthermore, R and C bits are maintained only for accesses made while address translation is enabled (MSR[IR] = 1 or MSR[DR] = 1).

In the e300 core, the reference and change bits are updated as follows:

- For TLB hits, the C bit is updated according to Table 6-7.
- For TLB misses, when a table search operation is in progress to locate a PTE, the R and C bits are updated (set, if required) to reflect the status of the page based on this access.

Table 6-7 shows that the status of the C bit in the TLB entry (in the case of a TLB hit) is what causes the processor to update the C bit in the PTE (the R bit is assumed to be set in the page tables if there is a TLB hit). Therefore, when software clears the R and C bits in the page tables in memory, it must invalidate the TLB entries associated with the pages whose reference and change bits were cleared.

Table 6-7. Table Search Operations to Update History Bits—TLB Hit Case

R and C Bits in TLB Entry	Processor Action	
00	Combination does not occur	
01	Combination does not occur	
10	Read: Write:	No special action Table search operation required to update C. Causes a data TLB miss on store interrupt.
11	No special action for read or write	

The core causes the R bit to be set for the execution of the **dcbt** or **dcbtst** instruction to that page (by causing a TLB miss interrupt to load the TLB entry in the case of a TLB miss). However, neither of these instructions causes the C bit to be set.

As defined by the PowerPC architecture, the reference and change bits are updated as if address translation were disabled (real addressing mode translation). Additionally, these updates should be performed with single-beat read and byte write transactions on the bus.

6.4.1.1 Reference Bit

The reference (R) bit of a page is located in the PTE in the page table. Every time a page is referenced (with a read or write access) and the R bit is zero, the R bit is then set in the page table. The OEA specifies that the reference bit may be set immediately, or the setting may be delayed until the memory access is determined to be successful. Because the reference to a page is what causes a PTE to be loaded into the TLB, the reference bit in all core TLB entries is effectively always set. The processor never automatically clears the reference bit.

The reference bit is only a hint to the operating system about the activity of a page. At times, the reference bit may be set by software although the access was not logically required by the program, or even if the access was prevented by memory protection. Examples of this in these systems include the following:

- Fetching of instructions not subsequently executed
- Accesses generated by an **lswx** or **stswx** instruction with a zero length
- Accesses generated by a **stwcx.** instruction when no store is performed because a reservation does not exist
- Accesses that cause interrupts and are not completed

6.4.1.2 Change Bit

The change bit of a page is located both in the PTE in the page table and in the copy of the PTE loaded into the TLB (if a TLB is implemented, as in the e300 core). Whenever a data store instruction is executed successfully, if the TLB search (for page address translation) results in a hit, the change bit in the matching TLB entry is checked. If it is already set, the processor does not change the C bit. If the TLB change bit is 0, it is set and a table search operation is performed to also set the C bit in the corresponding PTE in the page table. The e300 core causes a data TLB miss on store interrupt for this case so that the software can perform the table search operation for setting the C bit. Refer to Section 6.5.2, “Implementation-Specific Table Search Operation,” for an example code sequence that handles these conditions.

The change bit (in both the TLB and PTE in the page tables) is set only when a store operation is allowed by the page memory protection mechanism and all conditional branches occurring earlier in the program have been resolved (such that the store is guaranteed to be in the execution path). Furthermore, the following conditions may cause the C bit to be set:

- The execution of an **stwcx.** instruction is allowed by the memory protection mechanism, but a store operation is not performed because no reservation exists.
- The execution of an **stswx** instruction is allowed by the memory protection mechanism, but a store operation is not performed because the specified length is zero.
- The store operation is not performed because an interrupt occurs before the store is performed.

Again, note that although the execution of the **dcbt** and **dcbtst** instructions may cause the R bit to be set, they never cause the C bit to be set.

6.4.1.3 Scenarios for Reference and Change Bit Recording

This section provides a summary of the model (defined by the OEA) that is used by the processors for maintaining the reference and change bits. In some scenarios, the bits are guaranteed to be set by the processor, in some scenarios, the architecture allows that the bits may be set (not absolutely required), and in some scenarios, the bits are guaranteed to not be set.

In implementations that do not maintain the R and C bits in hardware (such as the e300 core), software assistance is required. For these processors, the information in this section still applies, except that the software performing the updates is constrained to the rules described (that is, must set bits shown as guaranteed to be set and must not set bits shown as guaranteed to not be set).

Table 6-8 defines a prioritized list of the R and C bit settings for all scenarios. The entries in the table are prioritized from top to bottom, such that a matching scenario occurring closer to the top of the table takes precedence over a matching scenario closer to the bottom of the table. For example, if an **stwcx.** instruction causes a protection violation and there is no reservation, the C bit is not altered, as shown for the protection violation case. Note that in the table, load operations include those generated by load instructions, by the **eciwx** instruction, and by the cache management instructions that are treated as a load with respect to address translation. Similarly, store operations include those operations generated by store instructions, by the **ecowx** instruction, and by the cache management instructions that are treated as a store with respect to address translation. Note that the e300 core does not support the **eciwx** or **ecowx** instructions, which are optional in the PowerPC architecture. In the columns for the e300 core, the combination of the core itself and the software used to search the page tables (described in Section 6.5.2, “Implementation-Specific Table Search Operation”) is assumed.

Table 6-8. Model for Guaranteed R and C Bit Settings

Priority	Scenario	R Bit Set		C Bit Set	
		OEA	G2 Core	OEA	G2 Core
1	No-execute protection violation	No	No	No	No
2	Page protection violation	Maybe	Yes	No	No
3	Out-of-order instruction fetch or load operation	Maybe	No	No	No
4	Out-of-order store operation for instructions that will cause no other kind of precise interrupt (in the absence of system-caused, imprecise, or floating-point assist interrupts[1]	Maybe[2]	No	No	No
5	All other out-of-order store operations	Maybe [1]	No	Maybe [1]	No
6	Zero-length load (**lswx**)	Maybe	Yes	No	No
7	Zero-length store (**stswx**)	Maybe [1]	Yes	Maybe [1]	Yes
8	Store conditional (**stwcx.**) that does not store	Maybe [1]	Yes	Maybe [1]	Yes
9	In-order instruction fetch	Yes[3]	Yes	No	No
10	Load instruction or **eciwx**[4]	Yes	Yes	No	No
11	Store instruction, **ecowx**[3] or **dcbz** instruction	Yes	Yes	Yes	Yes
12	**dcbt**, **dcbtst**, **dcbst**, or **dcbf** instruction	Maybe	Yes	No	No

Table 6-8. Model for Guaranteed R and C Bit Settings (continued)

Priority	Scenario	R Bit Set		C Bit Set	
		OEA	G2 Core	OEA	G2 Core
13	**icbi** instruction	Maybe [1]	No	No [1]	No
14	**dcbi**[5] instruction	Maybe [1]	Yes	Maybe [1]	Yes

1 Floating-point not supported on the e300c2.

2 If C is set, R is guaranteed to also be set.

3 This includes the case when the instruction was fetched out-of-order and R was not set (does not apply for the e300 core).

4 Not supported on the e300 core.

5 **The dcbi instruction should never be used on the** e300 **core.**

For more information, see “Page History Recording” in Chapter 7, “Memory Management,” of the *Programming Environments Manual*.

6.4.2 Page Memory Protection

The e300 core implements page memory protection as it is defined in Chapter 7, “Memory Management,” in the *Programming Environments Manual*.

6.4.3 TLB Description

This section describes the hardware resources provided in the e300 core to facilitate the page address translation process. Note that the hardware implementation of the MMU is not specified by the architecture, and while this description applies to the e300 core, it does not necessarily apply to other processors of this family.

6.4.3.1 TLB Organization

Because the e300 core has two MMUs (IMMU and DMMU) that operate in parallel, some of the MMU resources are shared, and some are actually duplicated (shadowed) in each MMU to maximize performance. Figure 6-7 shows the relationships between these resources within both the IMMU and DMMU and how the various portions of the effective address are used in the address translation process.

While both MMUs can be accessed simultaneously (both sets of segment registers and TLBs can be accessed in the same clock), when there is an exception condition, only one interrupt is reported at a time. ITLB miss interrupts are reported when there are no more instructions to be dispatched or retired (the pipeline is empty). Refer to Chapter 7, “Instruction Timing,” for more detailed information about the internal pipelines and the reporting of interrupts.

As TLB entries are on-chip copies of PTEs in the page tables in memory, they are similar in structure. TLB entries consist of two words; the high-order word contains the VSID and API fields of the high-order word of the PTE and the low-order word contains the RPN, C bit, WIMG bits, and PP bits (as in the low-order word of the PTE). In order to uniquely identify a TLB entry as the required PTE, the TLB entry also contains five more bits of the page index, EA[10–14] (in addition to the API bits of the PTE).

When an instruction or data access occurs, the effective address is routed to the appropriate MMU. EA[0–3] select 1 of the 16 segment registers and the remaining effective address bits and the virtual address from the segment register is passed to the TLB. EA[15–19] then select two entries in the TLB; the valid bit is checked and EA[10–14], VSID, and API fields (EA[4–9]) for the access are then compared with the corresponding values in the TLB entries. If one of the entries hits, the PP bits are checked for a protection violation, and the C bit is checked. If these bits do not cause an interrupt, the RPN value is passed to the memory subsystem and the WIMG bits are then used as attributes for the access.

Also, note that the segment registers do not have a valid bit, and so they should also be initialized before translation is enabled.

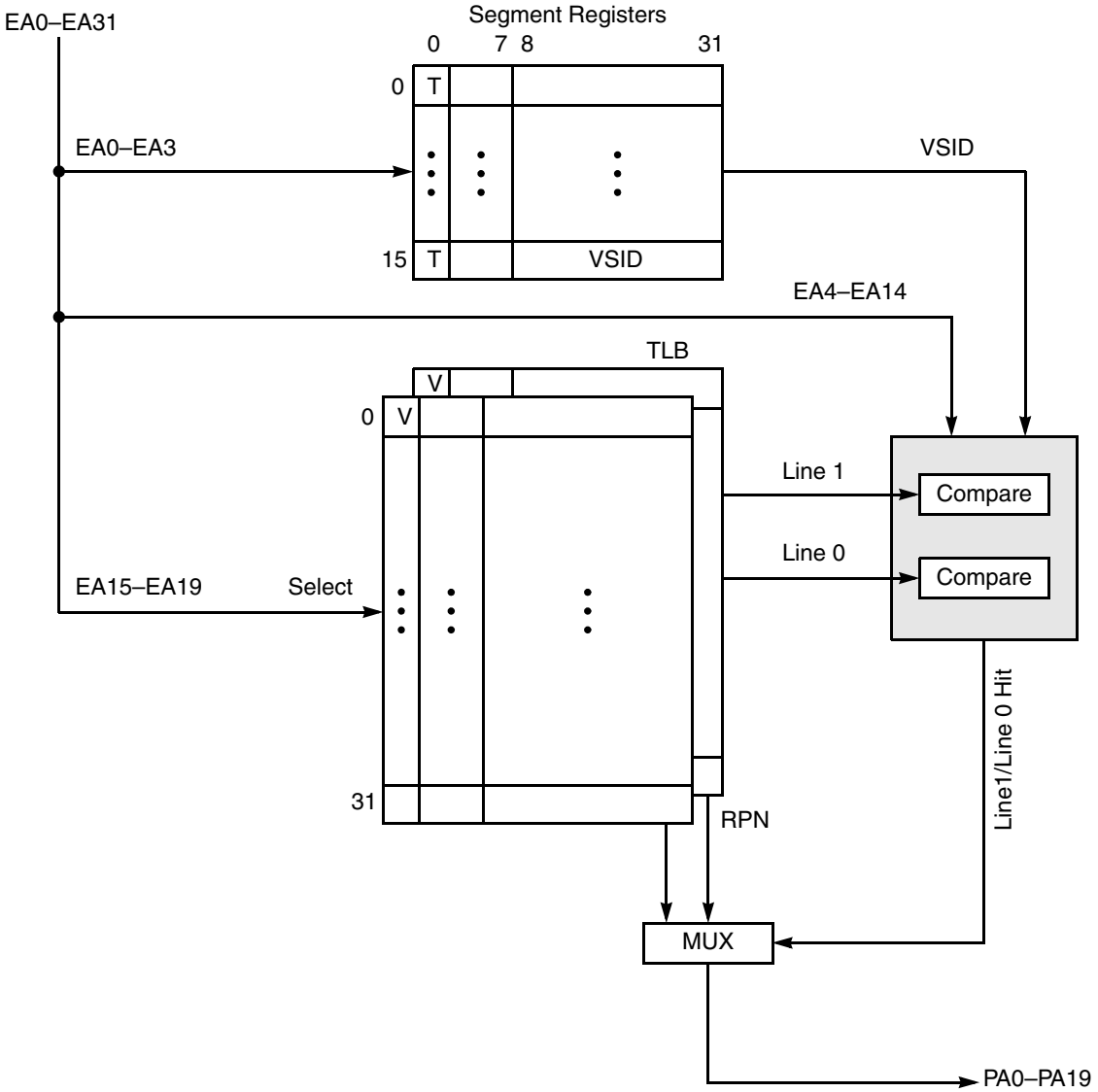


Figure 6-7. Segment Register and TLB Organization

6.4.3.2 TLB Entry Invalidation

For processors, such as the e300 core, that implement TLB structures to maintain on-chip copies of the PTEs that are resident in physical memory, the optional **tlbie** instruction provides a way to invalidate the TLB entries. Note that the execution of the **tlbie** instruction in the e300 core invalidates four entries—both the ITLB entries indexed by EA[15–19] and both the indexed entries of the DTLB.

The architecture allows **tlbie** to optionally enable a TLB invalidate signaling mechanism in hardware so that other processors also invalidate their resident copies of the matching PTE. The core does not signal the TLB invalidation to other processors and does not perform any action when a TLB invalidation is performed by another processor.

The **tlbsync** instruction causes instruction execution to stop if the $\overline{\textit{tlbisync}}$ input signal is also asserted. If $\overline{\textit{tlbisync}}$ is negated, instruction execution may continue or resume after the completion of a **tlbsync** instruction.

The **tlbia** instruction is not implemented in the e300 core and when its opcode is encountered, an illegal instruction program interrupt is generated. To invalidate all entries of both TLBs, 32 **tlbie** instructions must be executed, incrementing the value in EA[15–19] by 1 each time. See Chapter 8, “Instruction Set,” in the *Programming Environments Manual* for detailed information about the **tlbie** instruction.

6.4.4 Page Address Translation Summary

Figure 6-8 provides the detailed flow for the page address translation mechanism. The figure includes the checking of the N bit in the segment descriptor and then expands on the TLB Hit branch of Figure 6-6. The detailed flow for the TLB Miss branch is described in Section 6.5.1, “Page Table Search Operation—Conceptual Flow.” Note that as in the case of block address translation, if the **dcbz** instruction is attempted to be executed either in write-through mode or as cache-inhibited (W = 1 or I = 1), the alignment interrupt is generated. The checking of memory protection violation conditions for page address translation is described in Chapter 7, “Memory Management,” in the *Programming Environments Manual* for 32-bit implementations.

6.5 Page Table Search Operation

As stated earlier, the operating system must synthesize the table search algorithm for setting up the tables. The core TLB miss interrupt handlers also use this algorithm (with the assistance of some hardware-generated values) to load TLB entries when TLB misses occur, as described in Section 6.5.2, “Implementation-Specific Table Search Operation.”

6.5.1 Page Table Search Operation—Conceptual Flow

The table search process for a processor of this family varies slightly for 64- and 32-bit implementations. The main differences are the address ranges and PTE formats specified. See the *Programming Environments Manual* for the PTE format. An outline of the page table search process performed by a 32-bit implementation is as follows:

1. The 32-bit physical address of the primary PTEG is generated as described in Chapter 7, “Memory Management,” in the *Programming Environments Manual* for 32-bit implementations.

2. The first PTE (PTE0) in the primary PTEG is read from memory. PTE reads should occur with an implied WIM memory/cache mode control bit setting of 0b001. Therefore, they are considered cacheable and burst in from memory and placed in the cache.

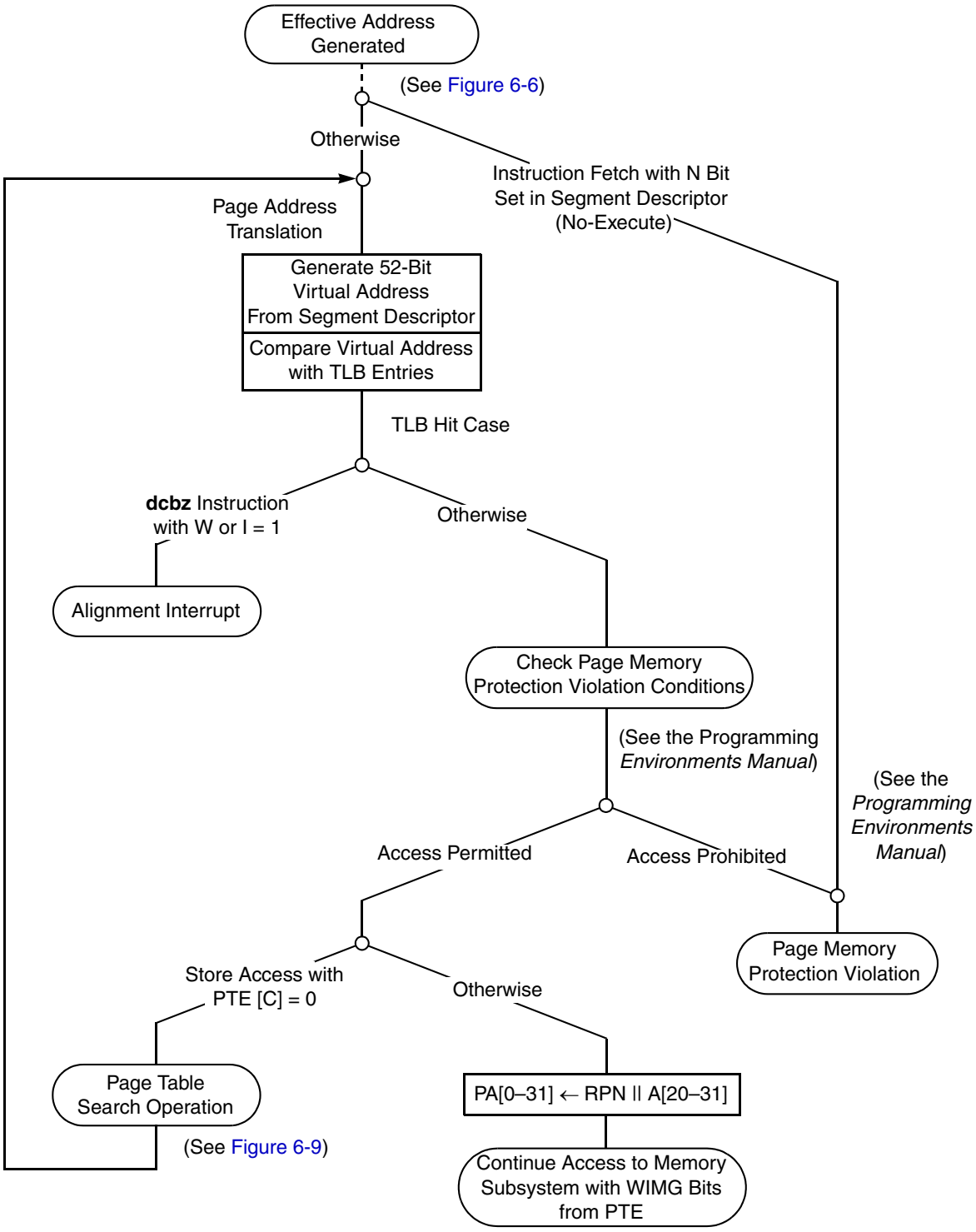

Effective Address Generated
(See Figure 6-6)
Otherwise
Instruction Fetch with N Bit Set in Segment Descriptor (No-Execute)
Page Address Translation
Generate 52-Bit Virtual Address From Segment Descriptor
Compare Virtual Address with TLB Entries
TLB Hit Case
dcbz Instruction with W or I = 1
Otherwise
Alignment Interrupt
Check Page Memory Protection Violation Conditions
(See the Programming Environments Manual)
(See the Programming Environments Manual)
Access Permitted
Access Prohibited
Page Memory Protection Violation
Store Access with PTE [C] = 0
Otherwise
Page Table Search Operation
(See Figure 6-9)
PA[0–31] ← RPN || A[20–31]
Continue Access to Memory Subsystem with WIMG Bits from PTE

Figure 6-8. Page Address Translation Flow for 32-Bit Implementations—TLB Hit

3. The PTE in the selected PTEG is tested for a match with the virtual page number (VPN) of the access. The VPN is the VSID concatenated with the page index field of the virtual address. For a match to occur, the following must be true:
   — PTE[H] = 0
   — PTE[V] = 1
   — PTE[VSID] = VA[0–23]
   — PTE[API] = VA[24–29]
4. If a match is not found, step 3 is repeated for each of the other seven PTEs in the primary PTEG. If a match is found, the table search process continues as described in step 8. If a match is not found within the eight PTEs of the primary PTEG, the address of the secondary PTEG is generated.
5. The first PTE (PTE0) in the secondary PTEG is read from memory. Again, because PTE reads typically have a WIM bit combination of 0b001, an entire cache line is burst into the on-chip cache.
6. The PTE in the selected secondary PTEG is tested for a match with the virtual page number (VPN) of the access. For a match to occur, the following must be true:
   — PTE[H] = 1
   — PTE[V] = 1
   — PTE[VSID] = VA[0–23]
   — PTE[API] = VA[24–29]
7. If a match is not found, step 6 is repeated for each of the other seven PTEs in the secondary PTEG.
8. If a match is found, the PTE is written into the on-chip TLB and the R bit is updated in the PTE in memory (if necessary). If there is no memory protection violation, the C bit is also updated in memory and the table search is complete.
9. If no match is found in the eight PTEs of the secondary PTEG, the search fails and a page fault interrupt condition occurs (either an ISI or DSI interrupt). Note that the software routines that implement this algorithm must synthesize this condition by appropriately setting the SRR1 or DSISR and branching to the ISI or DSI handler routine.

Reads from memory for table search operations should be performed as global (but not exclusive), cacheable operations, and can be loaded into the on-chip cache.

Figure 6-9 and Figure 6-10 provide conceptual flow diagrams of primary and secondary page table search operations as described in the OEA for 32-bit processors. Recall that the architecture allows implementations to perform the page table search operations automatically (in hardware) or with software assistance (may be required), as is the case with the e300 core. Also, the elements in the figure that apply to TLBs are shown as optional because TLBs are not required by the architecture.

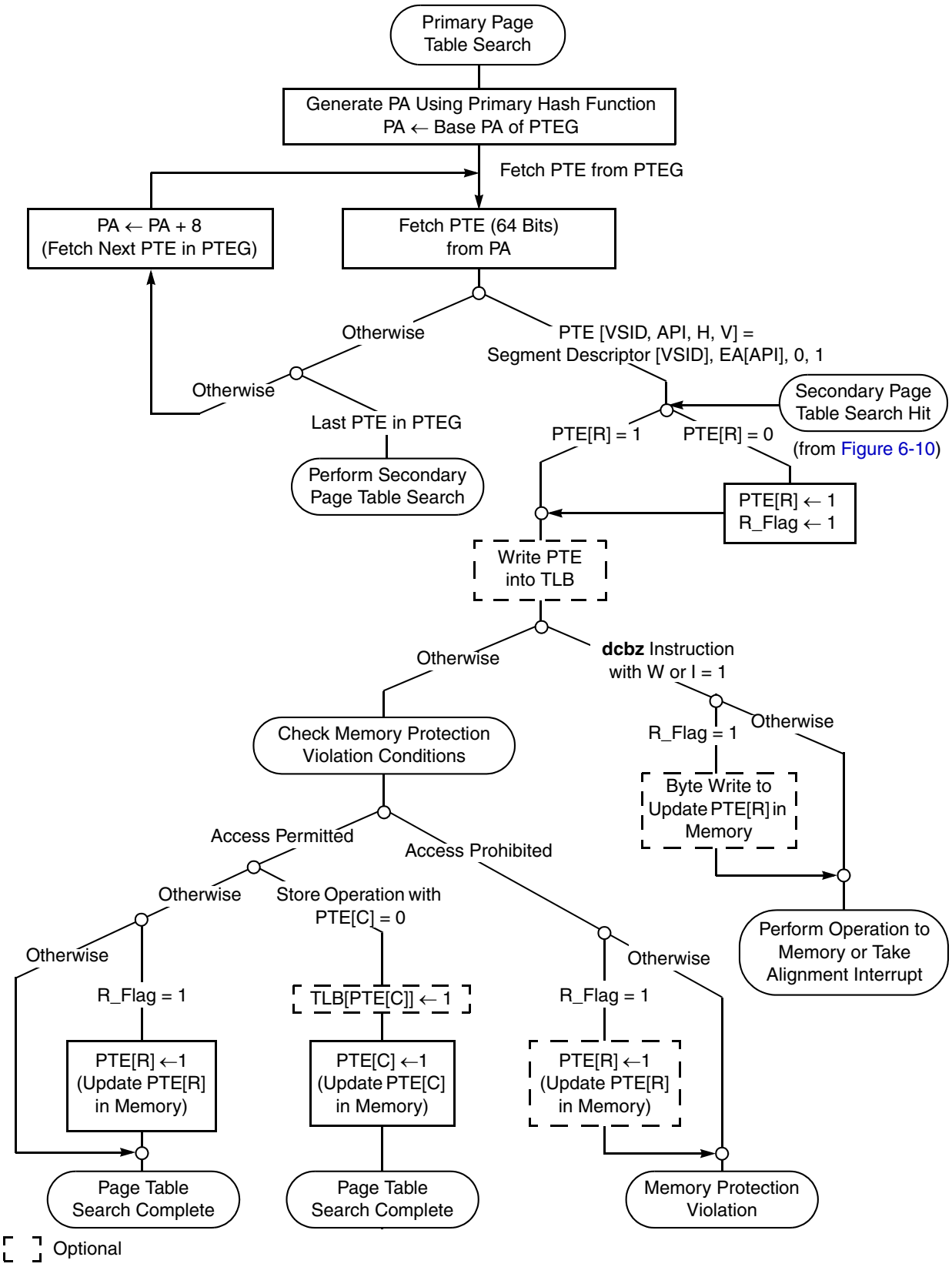

Primary Page Table Search
Generate PA Using Primary Hash Function
PA ← Base PA of PTEG
Fetch PTE from PTEG
PA ← PA + 8
(Fetch Next PTE in PTEG)
Fetch PTE (64 Bits) from PA
Otherwise
PTE [VSID, API, H, V] = Segment Descriptor [VSID], EA[API], 0, 1
Otherwise
Last PTE in PTEG
Secondary Page Table Search Hit
(from Figure 6-10)
PTE[R] = 1
PTE[R] = 0
Perform Secondary Page Table Search
PTE[R] ← 1
R_Flag ← 1
Write PTE into TLB
Otherwise
dcbz Instruction with W or I = 1
Check Memory Protection Violation Conditions
R_Flag = 1
Otherwise
Byte Write to Update PTE[R] in Memory
Access Permitted
Access Prohibited
Otherwise
Store Operation with PTE[C] = 0
Perform Operation to Memory or Take Alignment Interrupt
Otherwise
Otherwise
R_Flag = 1
TLB[PTE[C]] ← 1
R_Flag = 1
PTE[R] ←1 (Update PTE[R] in Memory)
PTE[C] ←1 (Update PTE[C] in Memory)
PTE[R] ←1 (Update PTE[R] in Memory)
Page Table Search Complete
Page Table Search Complete
Memory Protection Violation
Optional

Figure 6-9. Primary Page Table Search—Conceptual Flow

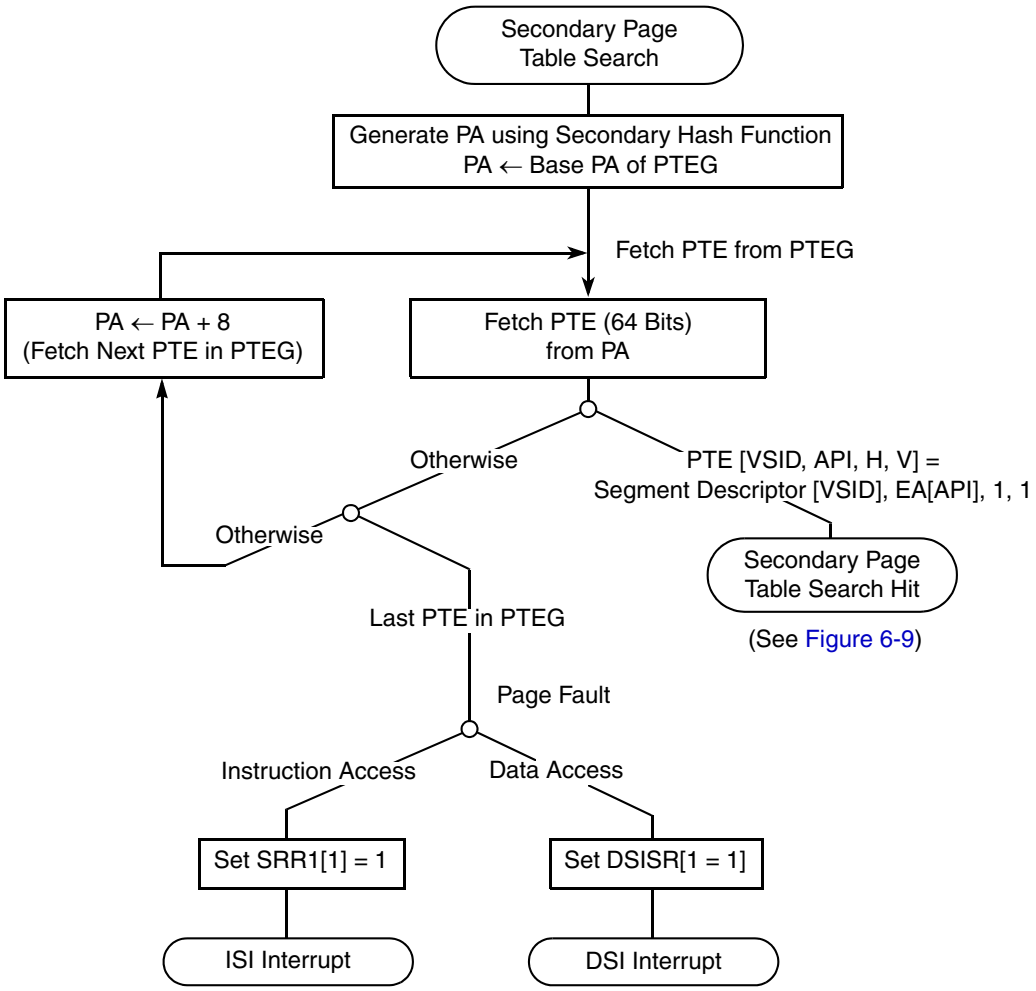


Figure 6-10. Secondary Page Table Search Flow—Conceptual Flow

Figure 6-9 shows the case of a **dcbz** instruction that is executed with W = 1 or I = 1, and that the R bit may be updated in memory (if required) before the operation is performed or the alignment interrupt occurs. The R bit may also be updated by a memory protection violation.

6.5.2 Implementation-Specific Table Search Operation

The e300 core has a set of implementation-specific registers, interrupts, and instructions that facilitate very efficient software searching of the page tables in memory. This section describes those resources and provides three example code sequences that can be used in an e300 core system for an efficient search of the translation tables in software. These three code sequences can be used as handlers for the three interrupts requiring access to the PTEs in the page tables in memory—instruction TLB miss, data TLB miss on load, and data TLB miss on store interrupts.

6.5.2.1 Resources for Table Search Operations

In addition to setting up the translation page tables in memory, the system software must assist the processor in loading PTEs into the on-chip TLBs. When a required TLB entry is not found in the

appropriate TLB, the processor vectors to one of the three TLB miss interrupt handlers so that the software can perform a table search operation and load the TLB. When this occurs, the processor automatically saves information about the access and the executing context. Table 6-9 provides a summary of the implementation-specific interrupts, registers, and instructions that can be used by the TLB miss interrupt handler software in e300 core systems. Refer to Chapter 5, "Interrupts and Exceptions," for more information about interrupt processing.

Table 6-9. Implementation-Specific Resources for Table Search Operations

Resource	Name	Description
Interrupts	Instruction TLB miss interrupt (vector offset 0x1000)	No matching entry found in ITLB
	Data TLB miss on load interrupt (vector offset 0x1100)	No matching entry found in DTLB for a load data access
	Data TLB miss on store interrupt—also caused when change bit must be updated (vector offset 0x1200)	No matching entry found in DTLB for a store data access or matching DLTB entry has C = 0 and access is a store
Registers	IMISS and DMISS	When a TLB miss interrupt occurs, IMISS or DMISS contains the 32-bit effective address of the instruction or data access that caused the miss interrupt.
	ICMP and DCMP	ICMP and DCMP contain the word to be compared with the first word of a PTE in the table search software routine to determine if a PTE contains the address translation for the instruction or data access. The contents of ICMP and DCMP are automatically derived by the core when a TLB miss interrupt occurs.
	HASH1 and HASH2	The HASH1 and HASH2 registers contain the primary and secondary PTEG addresses that correspond to the address causing a TLB miss. These PTEG addresses are automatically derived by the core by performing the primary and secondary hashing function on the contents of IMISS or DMISS, for an ITLB or DTLB miss interrupt, respectively.
	RPA	The system software loads a TLB entry by loading the second word of the matching PTE entry into the RPA register and then executing the **tlbli** or **tlbld** instruction (for loading the ITLB or DTLB, respectively).
Instructions	**tlbli r**B	Loads the contents of the ICMP and RPA registers into the ITLB entry selected by <ea> and SRR1[WAY]
	tlbld rB	Loads the contents of the DCMP and RPA registers into the DTLB entry selected by <ea> and SRR1[WAY]

In addition, the core contains the following features that do not specifically control the MMU, but that are implemented to increase performance and flexibility in the software table search routines whenever one of the three TLB miss interrupts occurs:

- Temporary GPR0–GPR3. These registers are available as **r0**–**r3** when MSR[TGPR] is set. The e300 core automatically sets MSR[TGPR] for these cases, allowing these interrupt handlers to have four registers that are used as scratchpad space, without having to save or restore this part of the machine state that existed when the interrupt occurred. Note that MSR[TGPR] is cleared when the **rfi** instruction is executed because the old MSR value (with MSR[TGPR] = 0) saved in SRR1 is restored. Refer to Section 6.5.2.2, "Software Table Search Operation," for code examples that take advantage of these registers.

- Also, the core automatically saves the values of CR[CR0] of the executing context to SRR1[0–3]. Thus, the interrupt handler can set CR[CR0] bits and branch accordingly in the interrupt handler routine, without having to save the existing CR[CR0] bits. However, the interrupt handler must restore these bits to CR[CR0] before executing the **rfi** instruction or branching to the DSI or ISI interrupt handler.

  In addition, SRR1[CRF0] must be cleared before branching to the DSI interrupt handler on a data access page fault. For an instruction access page fault, SRR1[0, 2–3] must be cleared before branching to the ISI handler. See Figure 6-17 for synthesizing a page fault interrupt when no PTE is found.
- SRR1[D/I] identifies an instruction or data miss, and SRR1[L/S] identifies a load or store miss. SRR1[WAY] identifies the associativity class of the TLB entry selected for replacement by the LRU algorithm. The software can change this value, effectively overriding the replacement algorithm. The SRR1[KEY] bit is used by the table search software to determine if there is a protection violation associated with the access (useful on data write misses for determining if the C bit should be updated in the table). Table 6-10 summarizes the SRR1 bits updated whenever one of the three TLB miss interrupts occurs.

Table 6-10. Implementation-Specific SRR1 Bits

Bits	Name	Function
0–3	CRF0	Condition register field 0 bits
12	KEY	Key for TLB miss (either Ks or Kp from segment register, depending on whether the access is a user or supervisor access).
13	D/I	Set if instruction TLB miss
14	WAY	Next TLB set to be replaced (set per LRU)
15	S/L	Set if data TLB miss was for a load instruction

The key bit saved in SRR1 is derived as follows:

Select KEY from segment register:

If MSR[PR] = 0, KEY = Ks

If MSR[PR] = 1, KEY = Kp

The rest of this section describes the format of the implementation-specific SPRs used by the TLB miss interrupt handlers. These registers can be accessed by supervisor-level instructions only. Because DMISS, IMISS, DCMP, ICMP, HASH1, HASH2, and RPA are used to access the translation tables for software table search operations, they should only be accessed when address translation is disabled (MSR[IR] = 0 and MSR[DR] = 0). Note that MSR[IR] and MSR[DR] are cleared whenever an interrupt occurs.

6.5.2.1.1 Data and Instruction TLB Miss Address Registers (DMISS and IMISS)

The DMISS and IMISS registers have the same format as shown in Figure 6-11. They are loaded automatically on a data or instruction TLB miss. The DMISS and IMISS contain the effective page address of the access which caused the TLB miss interrupt. The contents are used by the processor when calculating the values of HASH1 and HASH2, and by the **tlbld** and **tlbli** instructions when loading a new

TLB entry. Note that the core always loads a big-endian address into the DMISS register. These registers are both read- and write- accessible. However, great caution should be used when writing to these registers.

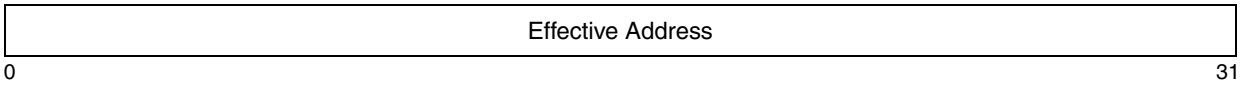


Figure 6-11. DMISS and IMISS Registers

6.5.2.1.2 Data and Instruction TLB Compare Registers (DCMP and ICMP)

The DCMP and ICMP registers are shown in Figure 6-12. These registers contain the first word in the required PTE. The contents are constructed automatically from the contents of the segment registers and the effective address (DMISS or IMISS) when a TLB miss interrupt occurs. Each PTE read from the tables in memory during the table search process should be compared with this value to determine whether or not the PTE is a match. Upon execution of a **tlbld** or **tlbli** instruction, the contents of the DCMP or ICMP register is loaded into the first word of the selected TLB entry.

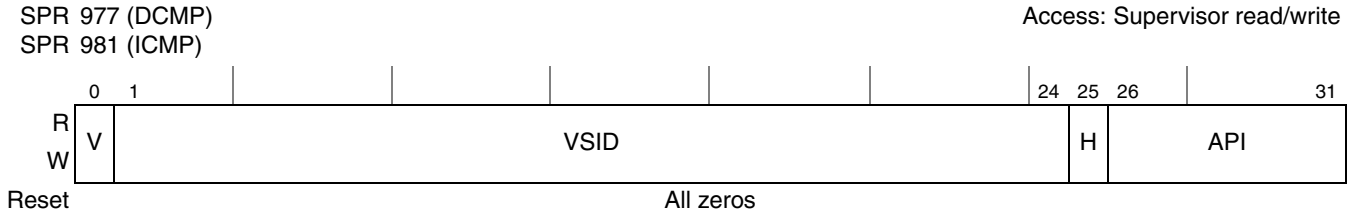


Figure 6-12. DCMP and ICMP Registers

Table 6-11 describes the bit settings for the DCMP and ICMP registers.

Table 6-11. DCMP and ICMP Bit Settings

Bits	Name	Description
0	V	Valid bit. Set by the processor on a TLB miss interrupt.
1–24	VSID	Virtual segment ID. Copied from VSID field of corresponding segment register.
25	H	Hash function identifier. Cleared by the processor on a TLB miss interrupt.
26–31	API	Abbreviated page index. Copied from API of effective address.

6.5.2.1.3 Primary and Secondary Hash Address Registers (HASH1 and HASH2)

HASH1 and HASH2 contain the physical addresses of the primary and secondary PTEGs for the access that caused the TLB miss interrupt. Only bits 7–25 differ between them. For convenience, the processor automatically constructs the full physical address by routing bits 0–6 of SDR1 into HASH1 and HASH2 and clearing the lower six bits. These registers are read-only and are constructed from the contents of the DMISS or IMISS register. The format for HASH1 and HASH2 is shown in Figure 6-13.

SPR 978 (HASH1)
SPR 979 (HASH2)

Access: Supervisor read-only

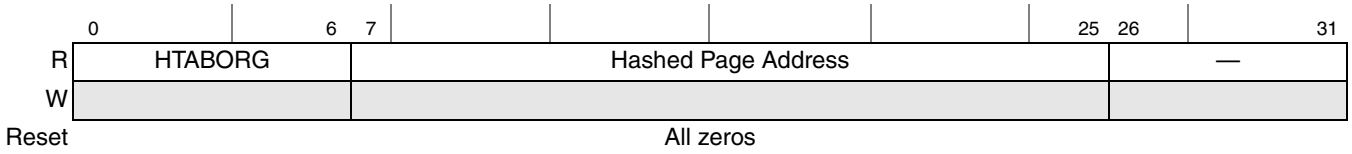


Figure 6-13. HASH1 and HASH2 Registers

Table 6-12 describes the bit settings of the HASH1 and HASH2 registers.

Table 6-12. HASH1 and HASH2 Bit Settings

Bits	Name	Description
0–6	HTABORG[0–6]	Copy of the upper 7 bits of the HTABORG field from SDR1
7–25	Hashed page address	Address bits 7–25 of the PTEG to be searched
26–31	—	Reserved

6.5.2.1.4 Required Physical Address Register (RPA)

The RPA is shown in Figure 6-14. During a page table search operation, the software must load the RPA with the second word of the correct PTE. When the **tlbld** or **tlbli** instruction is executed, data from IMISS and ICMP (or DMISS and DCMP) and the RPA registers is merged and loaded into the selected TLB entry. The TLB entry is selected by the effective address of the access (loaded by the table search software from the DMISS or IMISS register) and SRR1[WAY].

SPR 982

Access: Supervisor read/write

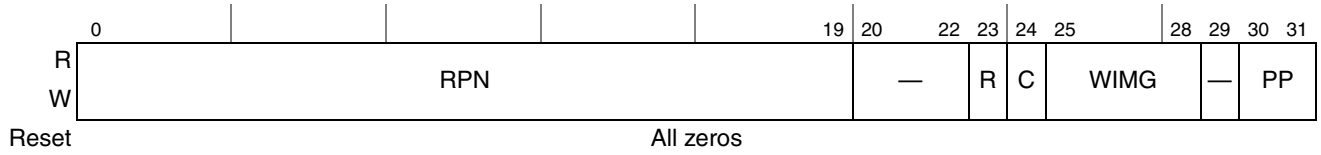


Figure 6-14. Required Physical Address Register (RPA)

Table 6-13 describes the bit settings of the RPA register.

Table 6-13. RPA Bit Settings

Bits	Name	Description
0–19	RPN	Physical page number from PTE
20–22	—	Reserved
23	R	Reference bit from PTE
24	C	Change bit from PTE
25–28	WIMG	Memory/cache access attribute bits
29	—	Reserved
30–31	PP	Page protection bits from PTE

6.5.2.2 Software Table Search Operation

When a TLB miss occurs, the instruction or data MMU loads IMISS or DMISS, with the effective address of the access. The processor completes all instructions ahead of the instruction that caused the interrupt, status information is saved in SRR1, and one of the three TLB miss interrupts is taken. In addition, the processor loads ICMP or DCMP with the value to be compared with the first word of PTEs in the tables in memory.

The software should then access the first PTE at the address pointed to by HASH1. The first word of the PTE should be loaded and compared to the contents of DCMP or ICMP. If there is a match, the required PTE has been found and the second word of the PTE is loaded from memory into RPA. Then the **tlbli** or **tlbld** instruction is executed, which loads the contents of ICMP or DCMP and RPA into the selected TLB entry. The TLB entry is selected by the effective address of the access and SRR1[WAY].

If the comparison does not match, the PTEG address is incremented to point to the next PTE in the table, and the above sequence is repeated. If none of the eight PTEs in the primary PTEG matches, the sequence is then repeated using the secondary PTEG (at the address contained in HASH2).

If the PTE is also not found in the eight entries of the secondary page table, a page fault condition exists and a page fault interrupt must be synthesized. Thus, the appropriate bits must be set in SRR1 (or DSISR) and the TLB miss handler must branch to either the ISI or DSI interrupt handler, which handles the page fault condition.

The following section provides a flow diagram outlining some example software that can be used to handle the three TLB miss interrupts and sample assembly language that implements that flow.

6.5.2.2.1 Flow for Example Interrupt Handlers

Figure 6-15 shows the flow for the example TLB miss interrupt handlers. The flow shown is common for the three interrupt handlers, except that the IMISS and ICMP registers are used for the instruction TLB miss interrupt while the DMISS and DCMP registers are used for the two data TLB miss interrupts. Also, for the cases of store instructions that cause either a TLB miss or require a table search operation to update the C bit, the flow shows that the C bit is set in both the TLB entry and PTE in memory. Finally, in the case of a page fault (no PTE found in the table search operation), the setup for the ISI or DSI interrupt is slightly different for these two cases.

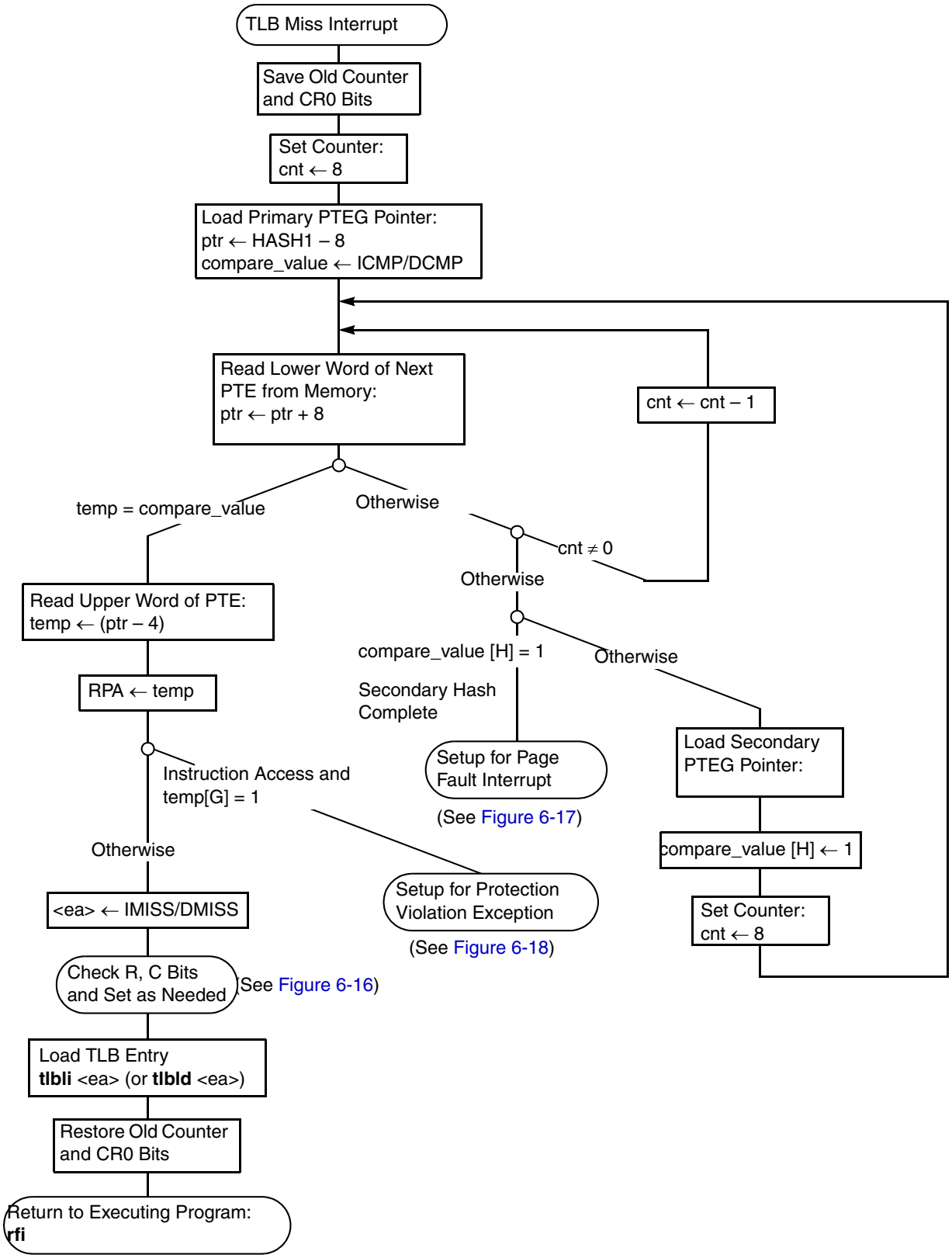

TLB Miss Interrupt
Save Old Counter and CR0 Bits
Set Counter: cnt ← 8
Load Primary PTEG Pointer: ptr ← HASH1 – 8 compare_value ← ICMP/DCMP
Read Lower Word of Next PTE from Memory: ptr ← ptr + 8
cnt ← cnt – 1
temp = compare_value
Otherwise
cnt ≠ 0
Otherwise
Read Upper Word of PTE: temp ← (ptr – 4)
compare_value [H] = 1
Otherwise
RPA ← temp
Secondary Hash Complete
Load Secondary PTEG Pointer:
Instruction Access and temp[G] = 1
Setup for Page Fault Interrupt
(See Figure 6-17)
Otherwise
compare_value [H] ← 1
Setup for Protection Violation Exception
(See Figure 6-18)
<ea> ← IMISS/DMISS
Set Counter: cnt ← 8
Check R, C Bits and Set as Needed
(See Figure 6-16)
Load TLB Entry tlbli <ea> (or tlbld <ea>)
Restore Old Counter and CR0 Bits
Return to Executing Program: rfi

Figure 6-15. Flow for Example Software Table Search Operation

The flow for checking the R and C bits and setting them appropriately is shown in Figure 6-16.

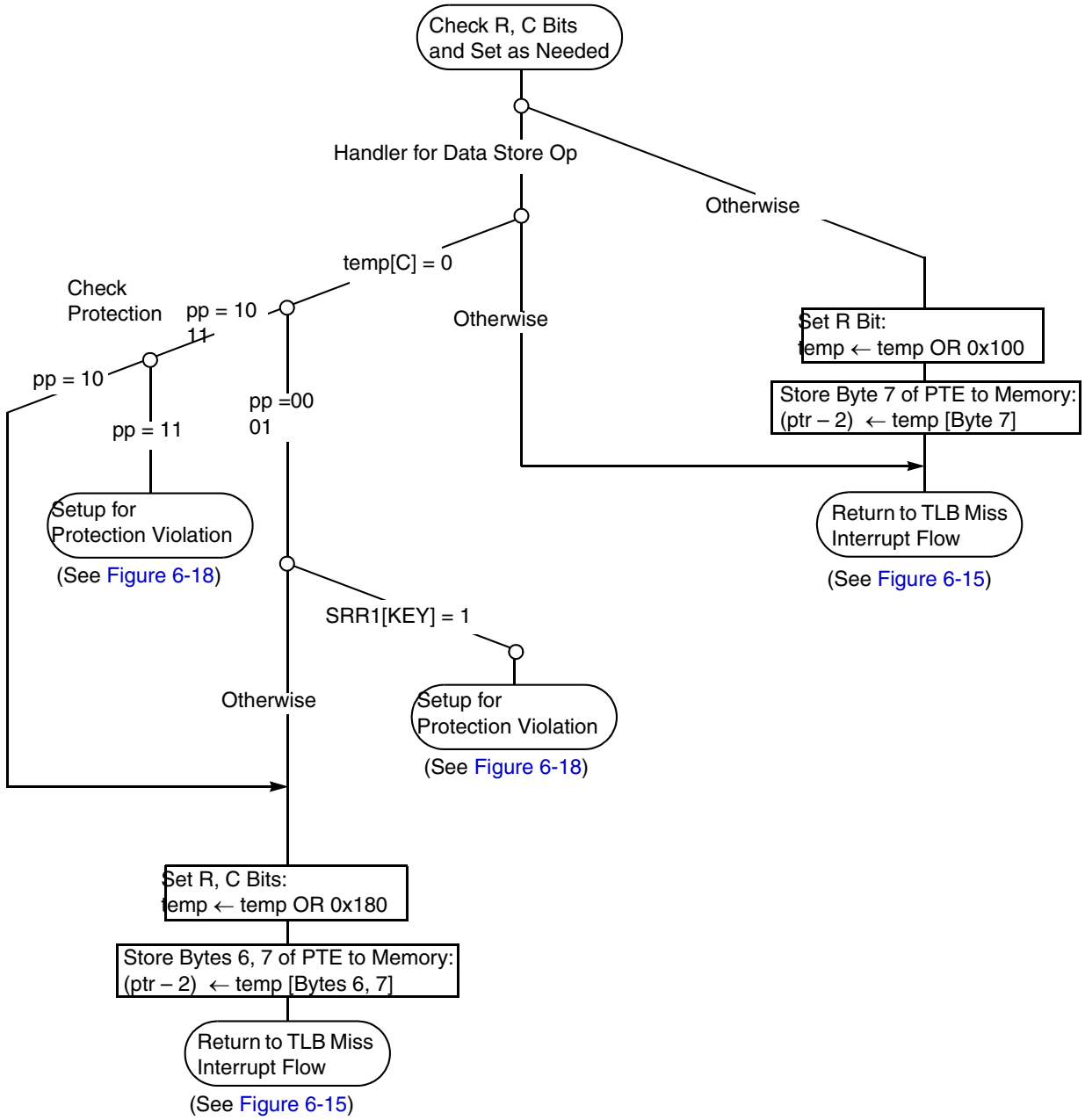


Figure 6-16. Check and Set R and C Bit Flow

Figure 6-17 shows the flow for synthesizing a page fault interrupt when no PTE is found.

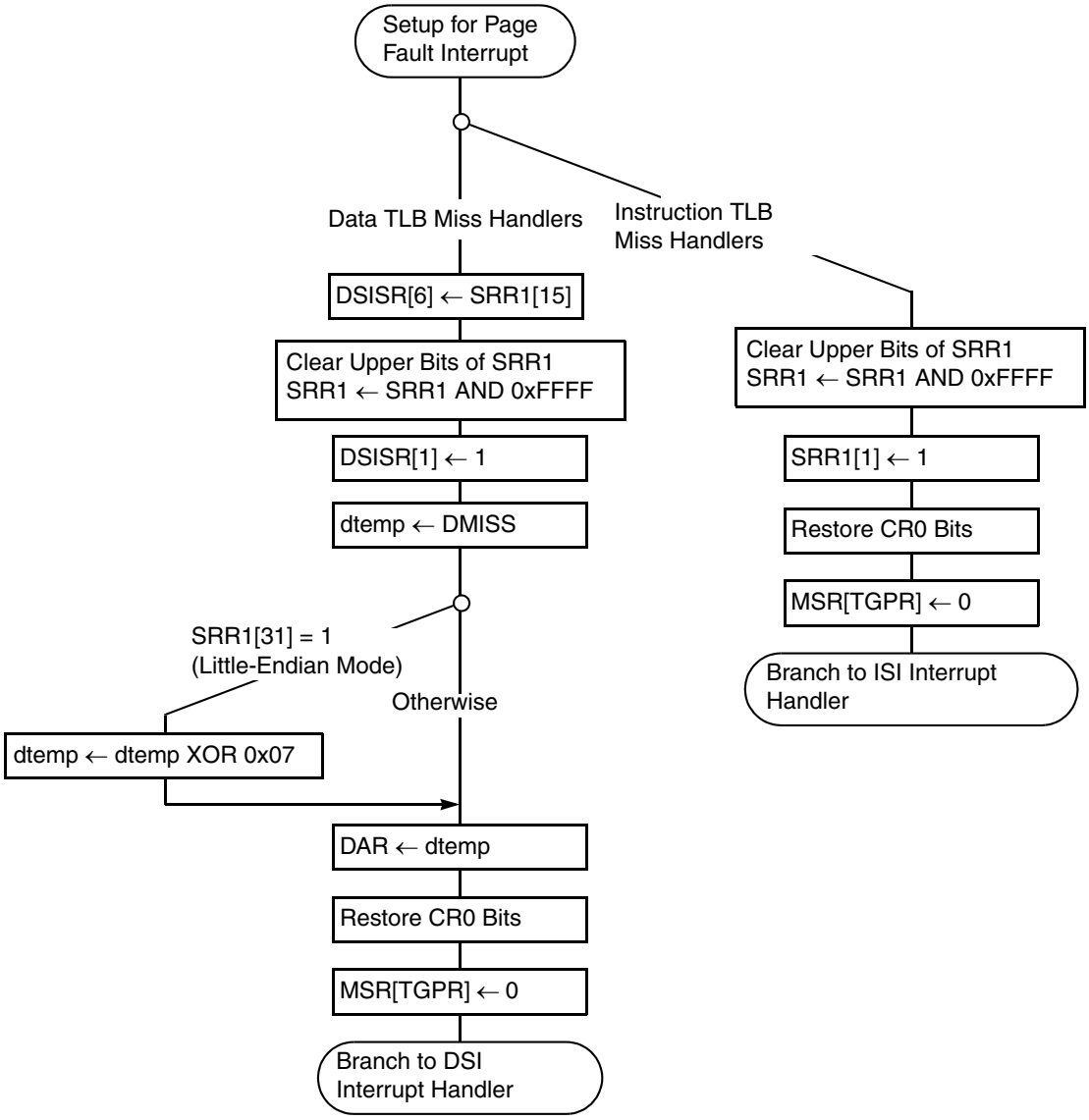


Figure 6-17. Page Fault Setup Flow

Figure 6-18 shows the flow for managing the cases of a TLB miss on an instruction access to guarded memory, and a TLB miss when C = 0 and a protection violation exists. The setup for these protection violation exceptions is very similar to that of page fault conditions (as shown in Figure 6-17) except that different bits in SRR1 (and DSISR) are set.

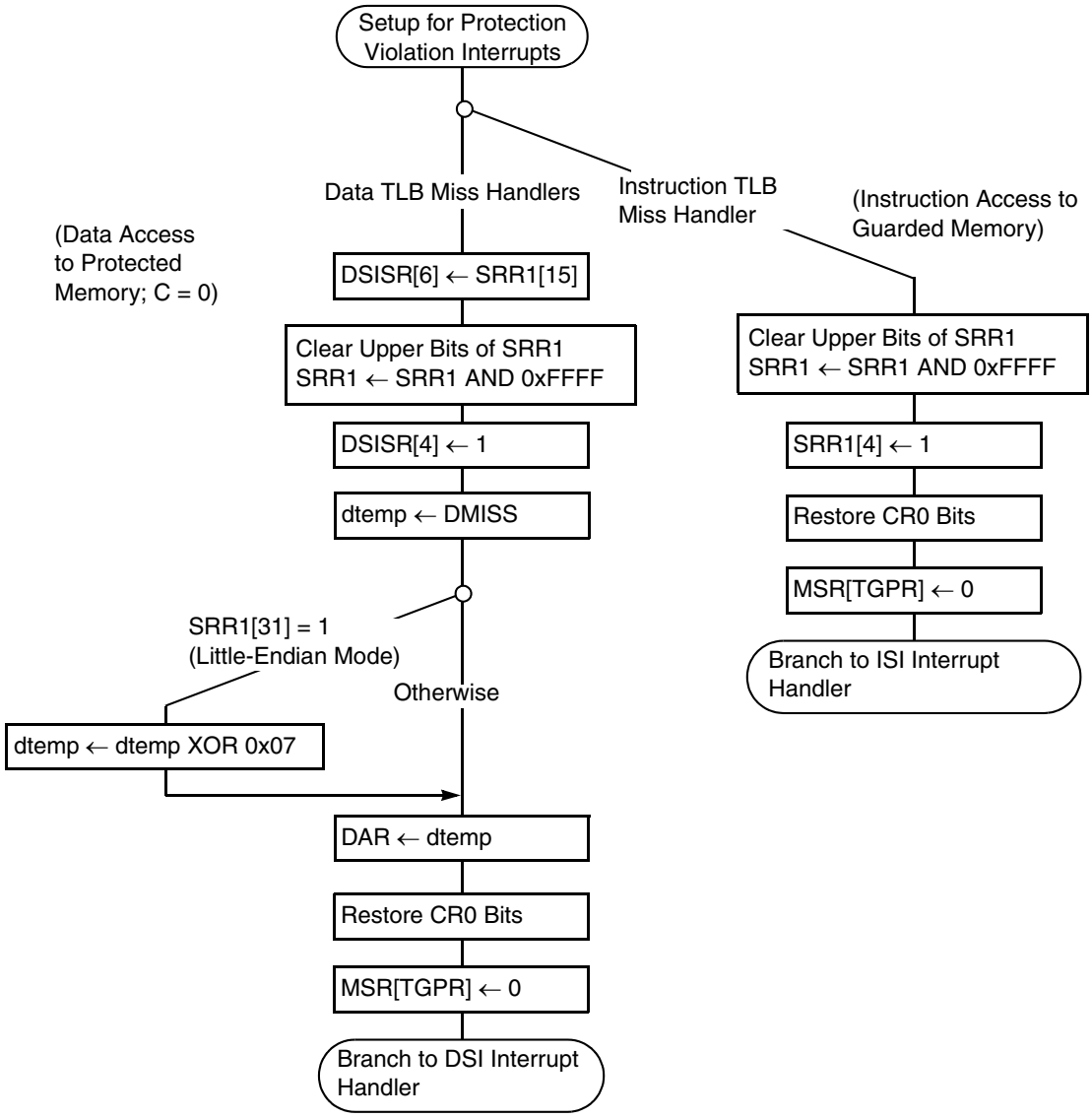


Figure 6-18. Setup for Protection Violation Exceptions

6.5.2.2.2 Code for Example Interrupt Handlers

This section provides assembly language examples that implement the flow diagrams described above. Note that although these routines fit into a few cache lines, they are supplied only as functional examples; they could be further optimized for faster performance.

```
# TLB software load for e300 core
#
# New Instructions:
#       tlbld                   - write the dtlb with the pte in rpa reg
```

```
#       tlbli                   - write the itlb with the pte in rpa reg
# New SPRs
#       dmiss                   - address of dstream miss
#       imiss                   - address of istream miss
#       hash1                   - address primary hash PTEG address
#       hash2                   - returns secondary hash PTEG address
#       iCmp                    - returns the primary istream compare value
#       dCmp                    - returns the primary dstream compare value
#       rpa                     - the second word of pte used by tlblx
#
# gpr r0..r3 are shadowed
#
# there are three flows.
#       tlbDataMiss       - tlb miss on data load
#       tlbCeq0           - tlb miss on data store or store with tlb change bit
                            == 0
#       tlbInstrMiss      - tlb miss on instruction fetch
#+
# place labels for rel branches
#-
#.machine e300
.set    r0, 0
.set    r1, 1
.set    r2, 2
.set    r3, 3
.set    dMiss, 976
.set    dCmp,    977
.set    hash1, 978
.set    hash2, 979
.set    iMiss, 980
.set    iCmp,   981
.set    rpa, 982
.set    c0, 0
.set    dar, 19
.set    dsisr, 18
.set    srr0, 26
.set    srr1, 27
.
.csect tlbmiss[PR]
vec0:
.globl vec0
.org    vec0+0x300
vec300:
.org    vec0+0x400
vec400:
#+
# Instruction TB miss flow
# Entry:
#       Vec = 1000
#       srr0       -> address of instruction that missed
#       srr1       -> 0:3=cr0 4=lru way bit 16:31 = saved MSR
#       msr<tgpr> -> 1
#       iMiss      -> ea that missed
#       iCmp       -> the compare value for the va that missed
#       hash1      -> pointer to first hash pteg
#       hash2      -> pointer to second hash pteg
#
```

```
# Register usage:
#       r0 is saved counter
#       r1 is junk
#       r2 is pointer to pteg
#       r3 is current compare value
.org    vec0+0x1000
tlbInstrMiss:
        mfspr   r2, hash1         # get first pointer
        addi    r1, 0, 8          # load 8 for counter
        mfctr   r0                # save counter
        mfspr   r3, iCmp          # get first compare value
        addi    r2, r2, -8        # pre dec the pointer
im0:    mtctr   r1                # load counter
im1:    lwzu    r1, 8(r2 )        # get next pte
        cmp     c0, r1, r3        # see if found pte
        bdnzf   eq, im1           # dec count br if cmp ne and if count not zero
        bne     instrSecHash              # if not found set up second hash or exit
        l       r1, +4(r2)                # load tlb entry lower-word
        andi.   r3, r1, 8                 # check G bit
        bne     doISIp                    # if guarded, take an ISI
        mtctr   r0                        # restore counter
        mfspr   r0, iMiss                 # get the miss address for the tlbli
        mfspr   r3, srr1                  # get the saved cr0 bits
        mtcrf   0x80, r3                  # restore CR0
        mtspr   rpa, r1                   # set the pte
        ori     r1, r1, 0x100             # set reference bit
        srwi    r1, r1, 8                 # get byte 7 of pte
        tlbli   r0                        # load the itlb
        stb     r1, +6(r2)                # update page table
        rfi                               # return to executing program
#+
# Register usage:
#       r0 is saved counter
#       r1 is junk
#       r2 is pointer to pteg
#       r3 is current compare value
#-
instrSecHash:
        andi.   r1, r3, 0x0040            # see if we have done second hash
        bne     doISI                     # if so, go to ISI interrupt
        mfspr   r2, hash2                 # get the second pointer
        ori     r3, r3, 0x0040            # change the compare value
        addi    r1, 0, 8                  # load 8 for counter
        addi    r2, r2, -8                # pre dec for update on load
        b       im0                       # try second hash
#+
# entry Not Found: synthesize an ISI interrupt
# guarded memory protection violation: synthesize an ISI interrupt
# Entry:
#       r0 is saved counter
#       r1 is junk
#       r2 is pointer to pteg
#       r3 is current compare value
#
doISIp:
        mfspr   r3, srr1                  # get srr1
        andi.   r2,r3,0xffff              # clean upper srr1
```

```
        addis   r2, r2, 0x0800          # or in srr<4> = 1 to flag prot violation
        b       isi1:
doISI:
        mfspr   r3, srr1                # get srr1
        andi.   r2, r3, 0xffff          # clean srr1
        addis   r2, r2, 0x4000          # or in srr1<1> = 1 to flag pte not found
isi1    mtctr   r0                      # restore counter
        mtspr   srr1, r2                # set srr1
        mfmsr   r0                      # get msr
        xoris   r0, r0, 0x8000          # flip the msr<tgpr> bit
        mtcrf   0x80, r3                # restore CR0
        mtmsr   r0                      # flip back to the native gprs
        b       vec400                  # go to instr. access interrupt
#
#+
# Data TLB miss flow
# Entry:
#       Vec = 1100
#       srr0       -> address of instruction that caused data tlb miss
#       srr1       -> 0:3=cr0 4=lru way bit 5=1 if store 16:31 = saved MSR
#       msr<tgpr> -> 1
#       dMiss      -> ea that missed
#       dCmp       -> the compare value for the va that missed
#       hash1      -> pointer to first hash pteg
#       hash2      -> pointer to second hash pteg
#
# Register usage:
#       r0 is saved counter
#       r1 is junk
#       r2 is pointer to pteg
#       r3 is current compare value
#-
.csect  tlbmiss[PR]
.org    vec0+0x1100
tlbDataMiss:
        mfspr   r2, hash1         # get first pointer
        addi    r1, 0, 8          # load 8 for counter
        mfctr   r0                # save counter
        mfspr   r3, dCmp          # get first compare value
        addi    r2, r2, -8        # pre dec the pointer
dm0:    mtctr   r1                # load counter
dm1:    lwzu    r1, 8(r2)         # get next pte
        cmp     c0, r1, r3        # see if found pte
        bdnzf   0, dm1            # dec count br if cmp ne and if count not zero
        bne     dataSecHash       # if not found set up second hash or exit
        l       r1, +4(r2)        # load tlb entry lower-word
        mtctr   r0                # restore counter
        mfspr   r0, dMiss         # get the miss address for the tlbld
        mfspr   r3, srr1          # get the saved cr0 bits
        mtcrf   0x80, r3          # restore CR0
        mtspr   rpa, r1           # set the pte
        ori     r1, r1, 0x100     # set reference bit
        srw     r1, r1, 8         # get byte 7 of pte
        tlbld   r0                # load the dtlb
        stb     r1, +6(r2)        # update page table
        rfi                       # return to executing program
```

```
#+
# Register usage:
#        r0 is saved counter
#        r1 is junk
#        r2 is pointer to pteg
#        r3 is current compare value
#-
dataSecHash:
        andi.    r1, r3, 0x0040    # see if we have done second hash
        bne      doDSI             # if so, go to DSI interrupt
        mfspr    r2, hash2         # get the second pointer
        ori      r3, r3, 0x0040    # change the compare value
        addi     r1, 0, 8          # load 8 for counter
        addi     r2, r2, -8        # pre dec for update on load
        b        dm0               # try second hash
#
#+
# C=0 in dtlb and dtlb miss on store flow
# Entry:
#        Vec = 1200
#        srr0       -> address of store that caused the interrupt
#        srr1       -> 0:3=cr0 4=lru way bit 5=1 16:31 = saved MSR
#        msr<tgpr> -> 1
#        dMiss      -> ea that missed
#        dCmp       -> the compare value for the va that missed
#        hash1      -> pointer to first hash pteg
#        hash2      -> pointer to second hash pteg
#
# Register usage:
#        r0 is saved counter
#        r1 is junk
#        r2 is pointer to pteg
#        r3 is current compare value
#-
.csect   tlbmiss[PR]
.org     vec0+0x1200
tlbCeq0:
        mfspr    r2, hash1         # get first pointer
        addi     r1, 0, 8          # load 8 for counter
        mfctr    r0                # save counter
        mfspr    r3, dCmp          # get first compare value
        addi     r2, r2, -8        # pre dec the pointer
ceq0:   mtctr    r1                # load counter
ceq1:   lwzu     r1, 8(r2)         # get next pte
        cmp      c0, r1, r3        # see if found pte
        bdnzf    0, ceq1           # dec count br if cmp ne and if count not zero
        bne      cEq0SecHash       # if not found set up second hash or exit
        l        r1, +4(r2)        # load tlb entry lower-word
        andi.    r3,r1,0x80        # check the C-bit
        beq      cEq0ChkProt       # if (C==0) go check protection modes
ceq2:   mtctr    r0                # restore counter
        mfspr    r0, dMiss         # get the miss address for the tlbld
        mfspr    r3, srr1          # get the saved cr0 bits
        mtcrf    0x80, r3          # restore CR0
        mtspr    rpa, r1           # set the pte
        tlbld    r0                # load the dtlb
        rfi                        # return to executing program
```

```
#+
# Register usage:
#        r0 is saved counter
#        r1 is junk
#        r2 is pointer to pteg
#        r3 is current compare value
#-
cEq0SecHash:
        andi.    r1, r3, 0x0040    # see if we have done second hash
        bne      doDSI             # if so, go to DSI interrupt
        mfspr    r2, hash2         # get the second pointer
        ori      r3, r3, 0x0040    # change the compare value
        addi     r1, 0, 8          # load 8 for counter
        addi     r2, r2, -8        # pre dec for update on load
        b        ceq0              # try second hash
#+
# entry found and PTE(c-bit==0):
# (check protection before setting PTE(c-bit)
# Register usage:
#        r0 is saved counter
#        r1 is PTE entry
#        r2 is pointer to pteg
#        r3 is trashed
#-
cEq0ChkProt:
        rlwinm.  r3,r1,30,0,1      # test PP
        bge-     chk0                      # if (PP==00 or PP==01) goto chk0:
        andi.    r3,r1,1                   # test PP[0]
        beq+     chk2                      # return if PP[0]==0
        b        doDSIp            # else DSIp
chk0:   mfspr    r3,srr1           # get old msr
        andis.   r3,r3,0x0008      # test the KEY bit (SRR1-bit 12)
        beq      chk2              # if (KEY==0) goto chk2:
        b        doDSIp            # else DSIp
chk2:   ori      r1, r1, 0x180     # set reference and change bit
        sth      r1, 6(r2)         # update page table
        b        ceq2              # and back we go
#
#+
# entry Not Found: synthesize a DSI interrupt
# Entry:
#        r0 is saved counter
#        r1 is junk
#        r2 is pointer to pteg
#        r3 is current compare value
#
doDSI:
        mfspr    r3, srr1          # get srr1
        rlwinm   r1, r3, 9,6,6     # get srr1<flag> to bit 6 for load/store, zero
                                     rest
        addis    r1, r1, 0x4000    # or in dsisr<1> = 1 to flag pte not found
        b        dsi1:
doDSIp:
        mfspr    r3, srr1          # get srr1
        rlwinm   r1, r3, 9,6,6     # get srr1<flag> to bit 6 for load/store, zero
                                     rest
        addis    r1, r1, 0x0800    # or in dsisr<4> = 1 to flag prot violation
```

```
dsi1:    mtctr    r0               # restore counter
         andi.    r2, r3, 0xffff   # clear upper bits of srr1
         mtspr    srr1, r2         # set srr1
         mtspr    dsisr, r1        # load the dsisr
         mfspr    r1, dMiss        # get miss address
         rlwinm.  r2,r2,0,31,31    # test LE bit
         beq      dsi2:            # if little endian then:
         xor      r1,r1,0x07       # de-mung the data address
dsi2:    mtspr    dar, r1                  # put in dar
         mfmsr    r0                       # get msr
         xoris    r0, r0, 0x2              # flip the msr<tgpr> bit
         mtcrf    0x80, r3                 # restore CR0
         mtmsr    r0                       # flip back to the native gprs
         b        vec300                   # branch to DSI interrupt
```

6.5.3 Page Table Updates

TLBs are defined as noncoherent caches of the PTEs. TLB entries must be flushed explicitly with the TLB invalidate entry instruction (**tlbie**) whenever the corresponding PTE is modified. Because the core is intended primarily for uniprocessor environments, it does not provide coherency checking for TLBs between multiple processors. If the e300 core is used in a multiprocessor environment where TLB coherency is required, synchronization must be implemented in software.

Processors may write reference and change bits with unsynchronized, atomic byte store operations. Note that each V, R, and C bits reside in a distinct byte of a PTE. Therefore, extreme care must be taken to use byte writes when updating only one of these bits.

Explicitly altering certain MSR bits (using the **mtmsr** instruction), PTEs, or certain system registers, may have the side effect of changing the effective or physical addresses from which the current instruction stream is being fetched. This kind of side effect is defined as an implicit branch. Implicit branches are not supported and an attempt to perform one causes boundedly-undefined results. Therefore, PTEs must not be changed in a manner that causes an implicit branch. Chapter 2, “Register Set,” in the *Programming Environments Manual*, lists the possible implicit branch conditions that can occur when system registers and MSR bits are changed.

6.5.4 Segment Register Updates

Synchronization requirements for using the move to segment register instructions (**mtsr** and **mtsrin**) are described in “Synchronization Requirements for Special Registers and for Lookaside Buffers” in Chapter 2, “Register Set,” in the *Programming Environments Manual*.

Chapter 7
Instruction Timing

This chapter describes how the e300 core processor fetches, dispatches, and executes instructions and how it reports the results of instruction execution. It gives detailed descriptions of how the core execution units work, and how those units interact with other parts of the processor, such as the instruction fetching mechanism, register files, and caches. It gives examples of instruction sequences, showing potential bottlenecks and how to minimize their effects. Finally, it includes tables that identify the unit that executes each instruction implemented on the core, the latency for each instruction, and other information that is useful for the assembly language programmer.

7.1 Terminology and Conventions

This section provides an alphabetical glossary of terms used in this chapter. These definitions are provided as a review of commonly used terms and as a way to point out specific ways these terms are used in this chapter.

- Branch prediction—The process of guessing whether a branch will be taken. Such predictions can be correct or incorrect; the term predicted as it is used here does not imply that the prediction is correct (successful). The PowerPC architecture defines a means for static branch prediction as part of the instruction encoding.
- Branch resolution—The determination of whether a branch is taken or not taken. A branch is said to be resolved when the processor can determine which instruction path to take. If the branch is resolved as predicted**,** the instructions following the predicted branch that may have been speculatively executed can complete (see completion). If the branch is not resolved as predicted**,** instructions on the mispredicted path, and any results of speculative execution, are purged from the pipeline and fetching continues from the nonpredicted path.
- Completion—Completion occurs when an instruction has finished executing, written back any results, and is removed from the completion queue (CQ). When an instruction completes, it is guaranteed that this instruction and all previous instructions can cause no interrupts.
- Fall-through (branch fall-through)—A not-taken branch. On the e300 core, fall-through branch instructions are removed from the instruction stream at dispatch. That is, these instructions are allowed to fall through the instruction queue through the dispatch mechanism, without either being passed to an execution unit and or given a position in the CQ.
- Fetch—The process of bringing instructions from memory (such as a cache or system memory) into the instruction queue.
- Finish—Finishing occurs in the last cycle of execution. In this cycle, the CQ entry is updated to indicate that the instruction has finished executing.
- Folding (branch folding)—The replacement of a branch instruction with target instructions and any instructions along the not-taken path, when a branch is either taken or predicted as taken.
- Latency—The number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction.
- Pipeline—In the context of instruction timing**,** the term pipeline refers to the interconnection of the stages. The events necessary to process an instruction are broken into several cycle-length tasks to

allow work to be performed on several instructions simultaneously—analogous to an assembly line. As an instruction is processed, it passes from one stage to the next. When it does, the stage becomes available for the next instruction.

Although an individual instruction may take many cycles to complete (the number of cycles is called instruction latency), pipelining makes it possible to overlap the processing so that the throughput (number of instructions completed per cycle) is greater than if pipelining were not implemented.

- Program order—The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.
- Rename register—Temporary buffers used by instructions that have finished execution but have not completed.
- Reservation station—A buffer between the dispatch and execute stages that allows instructions to be dispatched even though the results of instructions on which the dispatched instruction may depend are not available.
- Retirement—Removal of the completed instruction from the CQ.
- Stage—The term stage is used in two different senses, depending on whether the pipeline is being discussed as a physical entity or a sequence of events. In the latter case, a stage is an element in the pipeline during which certain actions are performed, such as decoding the instruction, performing an arithmetic operation, or writing back the results. A stage is typically described as taking a processor clock cycle to perform its operation; however, some events (such as dispatch and write-back) happen instantaneously, and may be thought to occur at the end of the stage.

  An instruction can spend multiple cycles in one stage. An integer **multiply,** for example, takes multiple cycles in the execute stage. When this occurs, subsequent instructions may stall.

  In some cases, an instruction may also occupy more than one stage simultaneously, especially in the sense that a stage can be seen as a physical resource—for example, when instructions are dispatched they are assigned a place in the CQ at the same time they are passed to the execute stage. They can be said to occupy both the complete and execute stages in the same clock cycle.
- Stall—An occurrence when an instruction cannot proceed to the next stage.
- Store queue—Holds store operations that have not been committed to memory, resulting from completed or retired instructions.
- Superscalar—A superscalar processor is one that can dispatch multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can be in the same stage at the same time.
- Throughput—A measure of the number of instructions that are processed per cycle. For example, a series of double-precision floating-point multiply instructions has a throughput of one instruction per clock cycle.
- Write-back—Write-back (in the context of instruction handling) occurs when a result is written from the rename registers into the architectural registers (typically the GPRs and FPRs or the store queue).

7.2 Instruction Timing Overview

The e300 core design minimizes average instruction execution latency, the number of clock cycles it takes to fetch, decode, dispatch, and execute instructions and make the results available for a subsequent instruction. Some instructions, such as loads and stores, access memory and require additional clock cycles between the execute phase and the write-back phase. These latencies vary depending on whether the access is to cacheable or noncacheable memory, whether it hits in the L1 cache, whether the cache access generates a write-back to memory, whether the access causes a snoop hit from another device that generates additional activity, and other conditions that affect memory accesses.

The core implements many features to improve throughput, such as pipelining, superscalar instruction dispatch, branch folding, removal of fall-through branches, two-level speculative branch handling, and multiple execution units that operate independently and in parallel.

As an instruction of load/store and floating-point units passes from stage to stage in a pipelined system, the following instruction can follow through the stages as the former instruction vacates them, allowing several instructions to be processed simultaneously. While it may take several cycles for an instruction to pass through all the stages, when the pipeline has been filled, one instruction can complete its work on every clock cycle.

Figure 7-1 represents a generic pipelined execution unit.

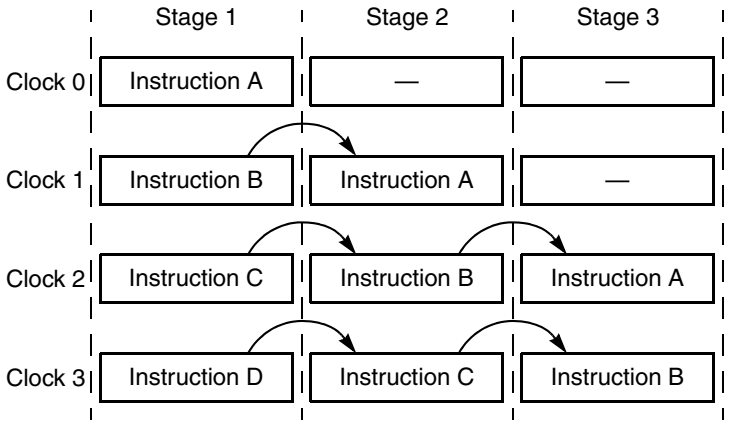


Figure 7-1. Pipelined Execution Unit

The entire path that instructions take through the fetch, decode/dispatch, execute, complete, and write-back stages is considered the e300 core master pipeline, and two of the core execution units (the FPU and LSU) are also multiple-stage pipelines.

The e300 core contains the following execution units that operate independently and in parallel:

- Branch processing unit (BPU)
- 32-bit integer unit (IU)—executes all integer instructions (dual integer units are supported on the e300c2, e300c3)
- 64-bit floating-point unit (FPU) (not supported on the e300c2 core)
- Load/store unit (LSU)
- System register unit (SRU)

The core can retire two instructions on every clock cycle. In general, the core processes instructions in four stages—fetch, decode/dispatch, execute, and complete as shown in Figure 7-2 for the e300c1, Figure 7-3 for the e300c2 core, and Figure 7-4 for the e300c3. Note that the example of a pipelined execution unit in Figure 7-1 is similar to the three-stage FPU pipeline in Figure 7-2.

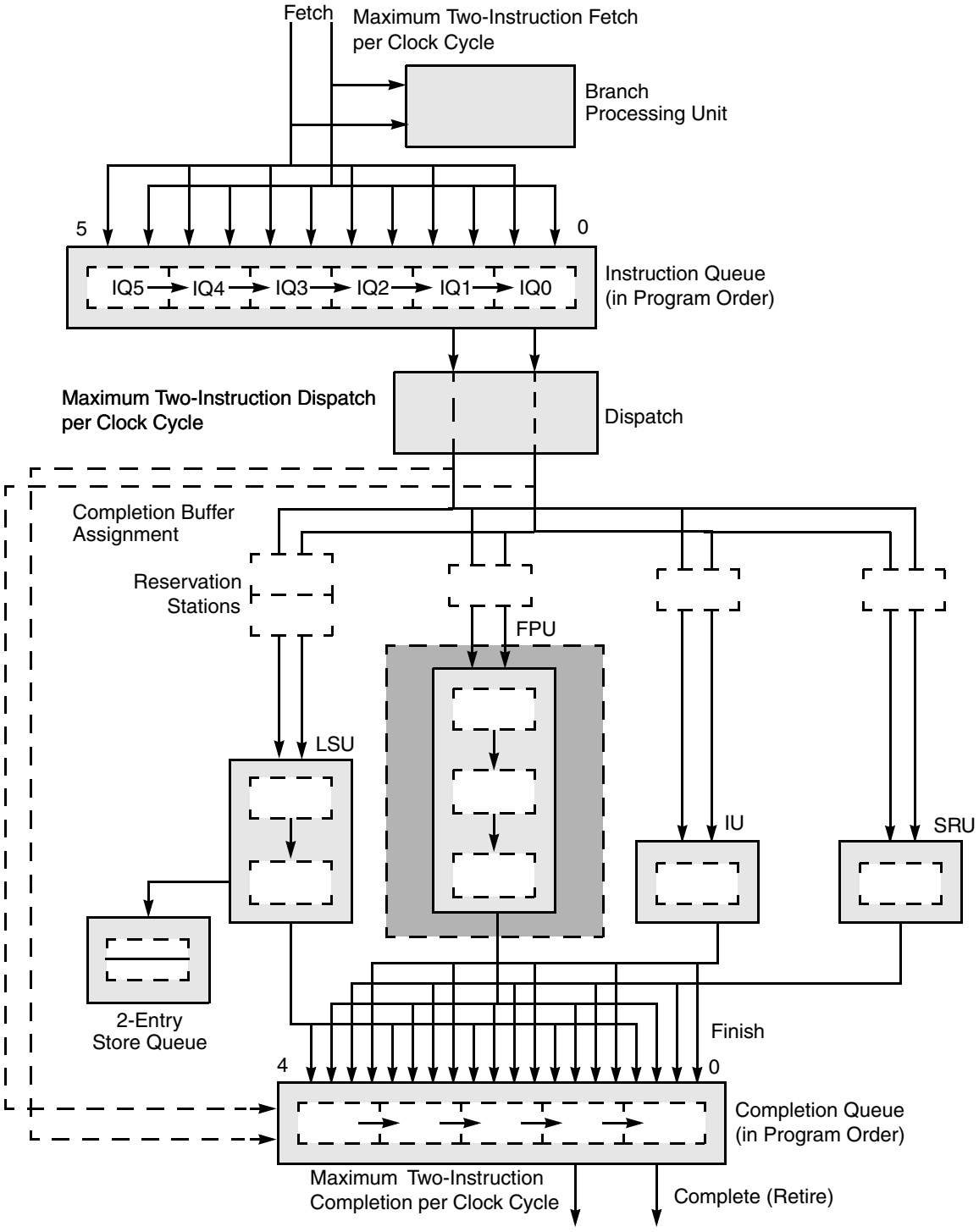


Figure 7-2. Instruction Flow Diagram for the e300c1

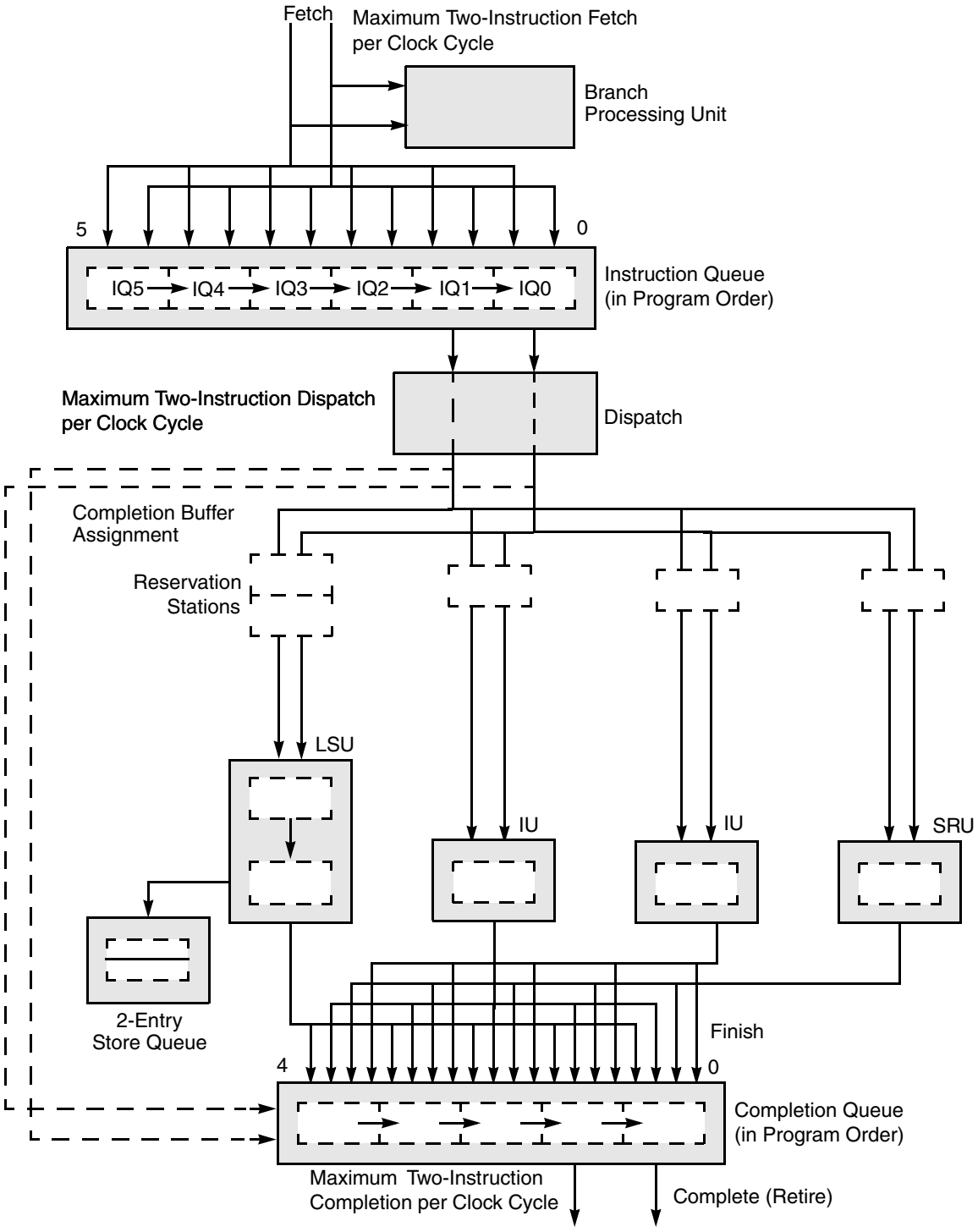


Figure 7-3. Instruction Flow Diagram for the e300c2

Figure 7-4 shows a block diagram of the e300c3. Note that the e300c3 support floating-point operations and includes two integer units.

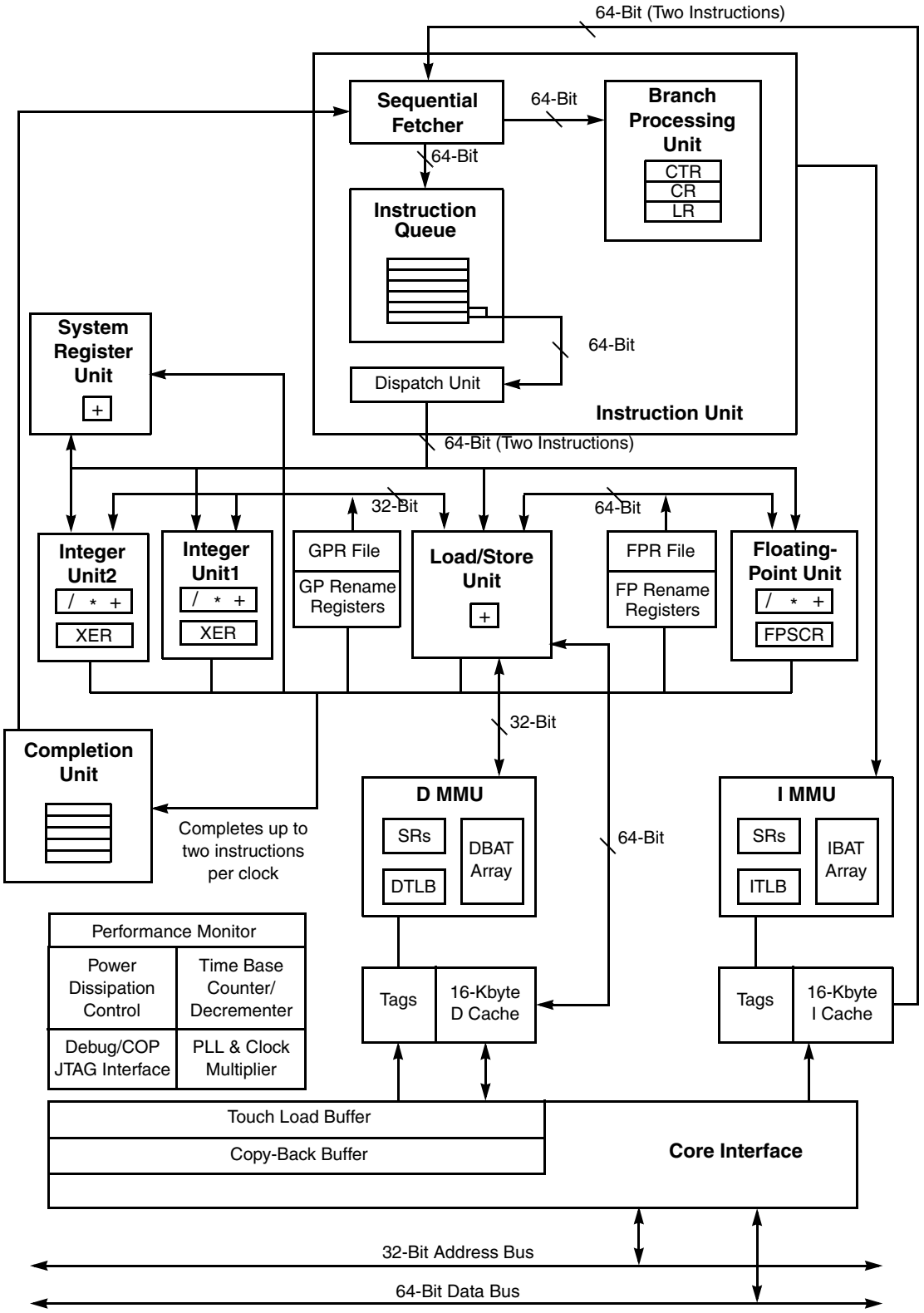


Figure 7-4. Instruction Flow Diagram for the e300c3

The instruction pipeline stages are described as follows:

- The instruction fetch stage includes the clock cycles necessary to request instructions from the memory system and the time the memory system takes to respond to the request. Instruction fetch timing depends on many variables, such as whether the instruction is in the branch target instruction cache, or in the on-chip instruction cache. Instruction fetch timing increases when it is necessary to fetch instructions from system memory. The variables that affect fetch timing include the processor-to-bus clock ratio, the amount of bus traffic, and whether any cache coherency operations are required.

  Because there are so many variables, unless otherwise specified, the instruction timing examples below assume optimal performance and that the instructions are available in the instruction queue in the same clock cycle that they are requested. The fetch stage ends when the instruction is dispatched.
- The decode/dispatch stage consists of the time it takes to fully decode the instruction and dispatch it from the instruction queue to the appropriate execution unit. Instruction dispatch requires the following:
  - Instructions can be dispatched only from the two lowest instruction queue entries, IQ0 and IQ1.
  - A maximum of two instructions can be dispatched per clock cycle.
  - Only one instruction can be dispatched to each execution unit per clock cycle.
  - There must be a vacancy in the specified execution unit.
  - A rename register must be available for each destination operand specified by the instruction.
  - For an instruction to dispatch, the appropriate execution unit must be available and there must be an open position in the CQ. If no entry is available, the instruction remains in the IQ.
- The execute stage consists of the time between dispatch to the execution unit (or reservation station) and the point at which the instruction vacates the execution unit.

  Most integer instructions have a one-cycle latency; results of these instructions can be used in the clock cycle after an instruction enters the execution unit. However, integer multiply and divide instructions take multiple clock cycles to complete. The IU can process all integer instructions.

  The e300c2, e300c3, integrate two integer units. The latency for multiply instructions in both units is now a maximum of 2 cycles to complete execution. Also, it is now possible to dispatch and execute two integer instruction types at one time (i.e. two multiply instructions), which significantly improves integer instruction throughput.
- The complete (complete/write-back) pipeline stage maintains the correct architectural machine state and commits it to the architectural registers at the proper time. If the completion logic detects an instruction containing an interrupt status, all following instructions are canceled, their execution results in rename registers are discarded, and the correct instruction stream is fetched.

  The complete stage ends when the instruction is retired. Two instructions can be retired per cycle. Instructions are retired only from the two lowest CQ entries, CQ0 and CQ1.

The notation conventions used in the instruction timing examples are as follows:

Fetch—The fetch stage includes the time between when an instruction is requested and when it is brought into the instruction queue. This latency can vary greatly, depending on whether the instruction is in the on-chip cache or system memory (in which case latency can be affected by bus speed and traffic on the system bus, and address translation dispatches). Therefore, in the examples in this chapter, the fetch stage is usually idealized; that is, an instruction is usually shown to be in the fetch stage when it is a valid instruction in the instruction queue. The instruction queue has six entries, IQ0–IQ5.

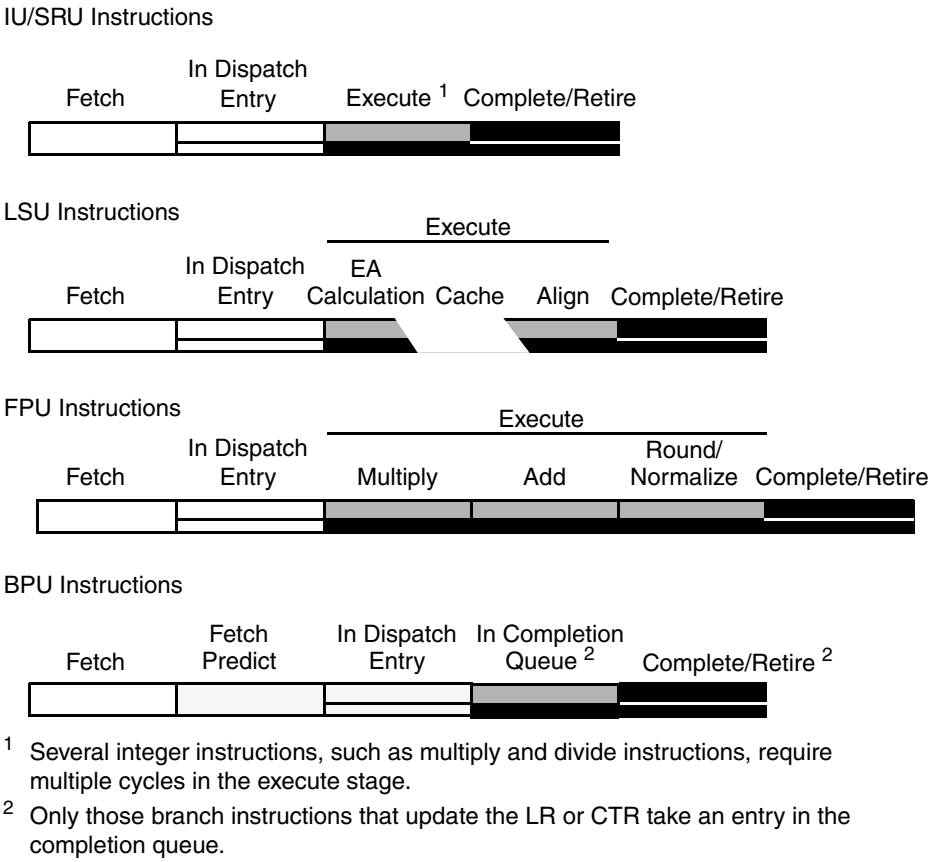
In dispatch entry (IQ0/IQ1)—Instructions can be dispatched from IQ0 and IQ1. Because dispatch is instantaneous, it is perhaps more useful to describe it as an event that marks the point in time between the last cycle in the fetch stage and the first cycle in the execute stage.

Execute—The operations specified by an instruction are being performed by the appropriate execution unit. The black stripe is a reminder that the instruction occupies an entry in the CQ, described in Figure 7-5.

Complete—The instruction is in the CQ. In the final stage, the results of the executed instruction are written back and the instruction is retired. The CQ has five entries, CQ0–CQ4.

In retirement entry—Completed instructions can be retired from CQ0 and CQ1. Like dispatch, retirement is an event that in this case occurs at the end of the final cycle of the complete stage.

Figure 7-5 shows the stages of the core execution units.



1 Several integer instructions, such as multiply and divide instructions, require multiple cycles in the execute stage.

2 Only those branch instructions that update the LR or CTR take an entry in the completion queue.

Figure 7-5. e300 Core Processor Pipeline Stages

7.3 Timing Considerations

The e300 core is a superscalar processor; as many as three instructions can be dispatched to the execution units (one branch instruction to the branch processing unit, and two instructions dispatched from the dispatch queue to the other execution units) during each clock cycle. Only one instruction can be dispatched to each execution unit.

Although instructions appear to the programmer to execute in program order, the core improves performance by executing multiple instructions at a time, using hardware to manage dependencies. When an instruction is dispatched, the register file provides the source data to the execution unit. The register files and rename register have sufficient bandwidth to allow dispatch of two instructions per clock under most conditions.

The BPU decodes and executes branches immediately after they are fetched. When a conditional branch cannot be resolved due to a CR data dependency, the branch direction is predicted and execution continues from the predicted path. If the prediction is incorrect, the following steps are taken:

1. The instruction queue is purged and fetching continues from the correct path.
2. Any instructions ahead of the predicted branch in the CQ are allowed to complete.
3. Instructions after the mispredicted branch are purged.
4. Dispatching resumes from the correct path.

After an execution unit executes an instruction, it places resulting data into the appropriate GPR or FPR rename register. The results are then stored into the correct GPR or FPR during the write-back stage. If a subsequent instruction needs the result as a source operand, it is made available simultaneously to the appropriate execution unit, which allows a data-dependent instruction to be decoded and dispatched without waiting to read the data from the register file. Branch instructions that update either the LR or CTR write back their results in a similar fashion.

The following section describes this process in greater detail.

7.3.1 General Instruction Flow

As many as two instructions can be fetched into the instruction queue (IQ) in a single clock cycle. Instructions enter the IQ and are dispatched to the various execution units from the dispatch queue. The IQ is a six-entry queue, which together with the CQ is the backbone of the master pipeline for the microprocessor. The core tries to keep the IQ full at all times.

The number of instructions requested in a clock cycle is determined by the number of vacant spaces in the IQ during the previous clock cycle. This is shown in the examples in this chapter. Although the IQ can accept as many as two new instructions in a single clock cycle and even if there are more than two spaces available on the current clock cycle, if only one IQ entry was vacant on the previous cycle, only one instruction is fetched. Typically, instructions are fetched from the on-chip instruction cache. If the instruction request hits in the on-chip instruction cache, it can usually present the first two instructions of the new instruction stream in the next clock cycle, giving enough time for the next pair of instructions to be fetched from the cache with no idle cycles. Instructions not in the instruction cache are fetched from system memory.

Branch instructions that do not update the LR or CTR are removed from the instruction stream either by branch folding or removal of fall-through branch instructions, as described in Section 7.4.1.1, “Branch Folding.” Branch instructions that update the LR or CTR are treated as if they require dispatch (even through they are not dispatched to an execution unit in the process). They are assigned a position in the CQ to ensure that the CTR and LR are updated sequentially.

All other instructions are dispatched from IQ0 and IQ1. The dispatch rate depends on the availability of resources such as the execution units, rename registers, and CQ entries, and on the serializing behavior of some instructions. Instructions are dispatched in program order; an instruction in IQ1 can be dispatched at the same time as one in IQ0, but cannot be dispatched ahead of one in IQ0.

Instruction state and all information required for completion is kept in the five-entry, FIFO completion queue. A completion queue entry is allocated for each instruction when it is dispatched to an execute unit; if no entry is available, the dispatch unit stalls. A maximum of two instructions per cycle may be completed and retired from the completion queue, and the flow of instructions can stall when a longer-latency instruction reaches the last position in the completion queue. Store instructions and instructions executed by the FPU and SRU (except for of integer add and compare instructions) can only be retired from the last position in the completion queue. Subsequent instructions cannot be completed and retired until that longer-latency instruction completes and retires. Examples of this are shown in Section 7.3.2.2, “Cache Hit,” and Section 7.3.2.3, “Cache Miss.”

The rate of instruction completion is also affected by the ability to write instruction results from the rename registers to the architected registers. The core can perform two write-back operations from the rename registers to the GPRs each clock cycle, but can perform only one write-back per cycle to the CR, FPR, LR, and CTR.

7.3.2 Instruction Fetch Timing

Instruction fetch latency depends on the fetch hits of the on-chip instruction cache. If no hit occurs, a memory transaction is required, in which case fetch latency is affected by bus traffic, bus clock speed, and memory translation. These conditions are discussed in the following sections.

7.3.2.1 Cache Arbitration

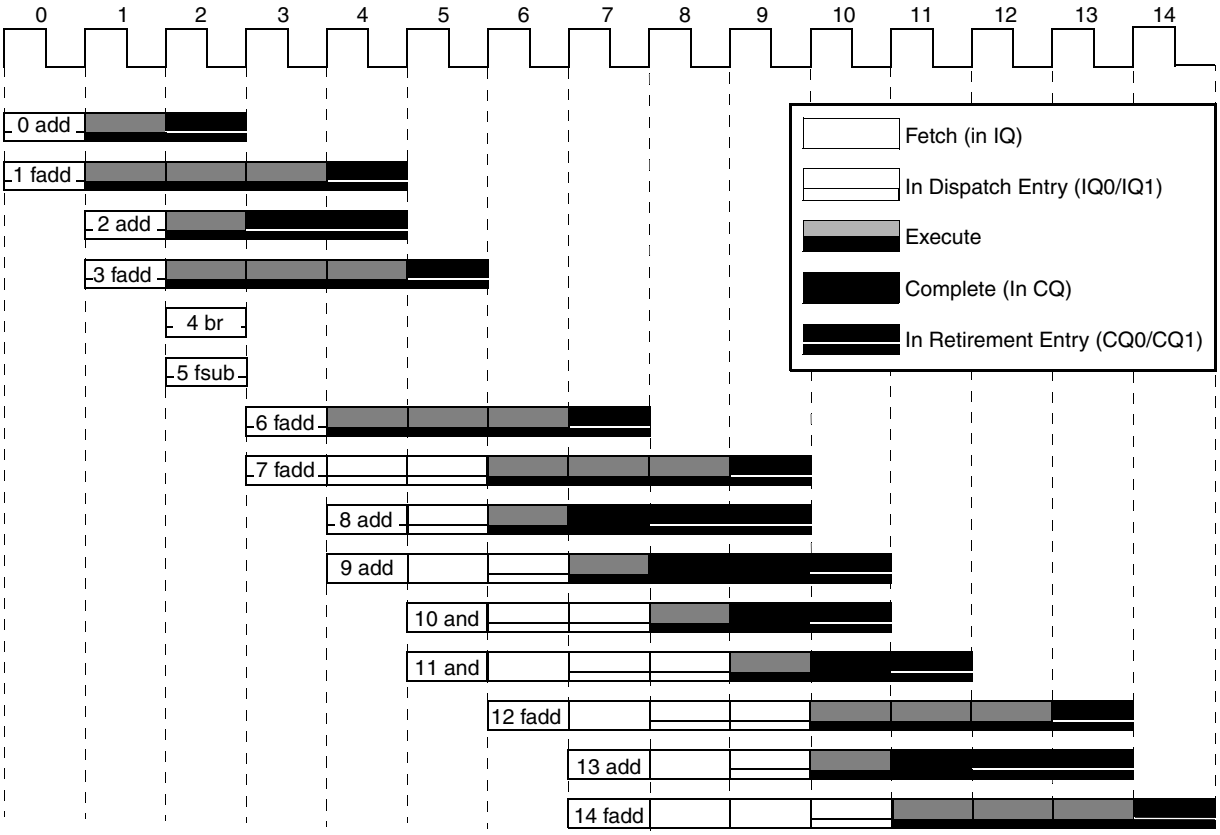
When the fetcher requests instructions from the cache, two things may happen. If the instruction cache is idle and the requested instructions are present, they are provided on the next clock cycle. The instruction fetch cancel extension allows a new instruction fetch to be issued to the cache or to the bus if a cancelled instruction fetch is pending or active on the bus. This is also called hit-under-cancel capability.

7.3.2.2 Cache Hit

An instruction fetch that hits the instruction cache takes only one clock cycle after the request for as many as two instructions to enter the IQ. Note that the cache is not blocked to internal accesses until a cache reload completes (hits under misses). The critical-double-word is written simultaneously to the cache and forwarded to the requesting unit, minimizing stalls due to load delays.

Figure 7-6 shows a simple example of instruction fetching that hits in the on-chip cache. This example uses a series of integer **add**, **and**, and double-precision floating-point add instructions to show how the number of instructions to be fetched is determined, how program order is maintained by the IQ and CQ, how instructions are dispatched and retired in pairs (maximum), and how the FPU pipeline functions.Note that floating-point instructions are not supported on the e300c2. The following instruction sequence is examined:

```
0   add
1   fadd
2   add
3   fadd
4   br 6
5   fsub
6   fadd
7   fadd
8   add
9   add
10  and
11  and
12  fadd
13  add
14  fadd
15  .
16  .
17  .
```



Instruction Queue

					11		14							
					10	12	13	14						
				9	9	11	12	13	14					
1	3	5	7	8	8	10	11	12	13					
0	2	4	6	7	7	9	10	11	12	14				

Completion Queue

									11	13				
		3		6			9	10	10	12	14			
		2	3	3		8	8	9	9	11	13	14	14	
	1	1	2	2	6	7	7	8	8	10	12	13	13	
	0	0	1	1	3	6	6	7	7	9	11	12	12	14

Figure 7-6. Instruction Timing—Cache Hit

The instruction timing for this example is described cycle-by-cycle as follows:

0. In cycle 0, instructions 0 and 1 are fetched from the instruction cache and are placed in the two entries in the instruction queue (IQ0 and IQ1), where they can be dispatched on the next clock cycle.
1. In cycle 1, instructions 0 and 1 are dispatched to the IU and FPU, respectively. Notice that for instructions to be dispatched, they must be assigned positions in the CQ. In this case, because the CQ is empty, instructions 0 and 1 take the two lowest CQ entries (CQ0 and CQ1). Instructions 2 and 3 are fetched from the instruction cache.
2. At least two IQ positions were available in the IQ in cycle 1, so in cycle 2, instructions 4 and 5 are fetched. Instruction 4 is a branch unconditional instruction that resolves immediately as taken. Because the branch is taken and does not update CTR or LR, it can be folded from the IQ. Instruction 0 completes, writes back its results, and vacates the CQ by the end of the clock cycle. Instruction 1 enters the second FPU execute stage, instruction 2 enters the single-stage IU, and instruction 3 is dispatched into the first FPU stage.
3. In cycle 3, target instructions 6 and 7 are fetched, replacing the folded **br** instruction 4 and instruction 5. Instruction 1 enters the last FPU execute stage, instruction 2 has executed but must remain in the CQ until instruction 1 completes. Note that it can make its results available to subsequent instructions, but cannot be removed from the CQ. Instruction 3 passes into the last FPU execute stage. Note that all three FPU stages are full. To allow for the potential need for denormalization, the dispatch logic prevents instruction 7 (**fadd**) from being dispatched in the next clock cycle.
4. In cycle 4, target instructions (8 and 9) are fetched. Instruction 1 completes in cycle 4, allowing instruction 2, which had finished executing in the previous clock cycle, to be removed from the CQ. Instruction 6 replaces instruction 3 in the first stage of the FPU. Also, as will be shown in cycle 5, a single-cycle stall occurs when the FPU pipeline is full.
5. In cycle 5, instruction 3 completes, instruction 6 continues through the FPU pipeline, and although the first stage of the FPU pipeline is free, instruction 7 cannot be dispatched because of the potential need for one of the previous floating-point instructions to require denormalization. Because instruction 7 cannot be dispatched neither can instruction 8. This dispatch stall causes the instruction queue to become full when instructions 10 and 11 are fetched.
6. In cycle 6, instruction 12 is fetched. Instruction 7 is dispatched to the first FPU stage, so instruction 8 can also be dispatched to the IU. Instructions 9 and 10 move to IQ0 and IQ1, but because instructions 9, 10, and 11 are integer instructions, only one instruction is dispatched in each of the next two clock cycles. Note that moving instruction 12 (**fadd**) up further in the program flow would improve dispatch throughput.
7. In cycle 7, instruction 6 completes, instruction 7 is in the second FPU execute stage, and although instruction 8 has executed, it must wait for instruction 7 to complete. Instruction 9 dispatches to the IU. Instructions 10 and 11 move down in the IQ. Fetching resumes with instructions 13 and 14.
8. In cycle 8, instruction 7 is in the third FPU execute stage. Instructions 8 and 9 have executed and they remain in the CQ until instruction 7 completes. Instruction 10 is dispatched to the IU.
9. In cycle 9, instruction 7 completes, allowing instruction 8 to complete. Because the CQ is full, instructions 12 and 13 cannot be dispatched.

10. In cycle 10, instructions 9 and 10 complete. Instruction 11 has executed but cannot exit the CQ from CQ2. Instructions 12 and 13 are dispatched to the FPU and IU, respectively. Instruction 14 drops into IQ0.
11. In cycle 11, instruction 11 completes and instruction 12 is in the second FPU execute stage. Instruction 13 has executed but must remain in the CQ until instruction 12 completes. Instruction 14 enters the first FPU execute stage.

7.3.2.3 Cache Miss

Figure 7-7 shows an instruction fetch that misses the on-chip cache and shows how that fetch affects the instruction dispatch. Note that a processor/bus clock ratio of 1:2 is used. The same instruction sequence is used as in Section 7.3.2.2, “Cache Hit.”

A cache miss extends the latency of the fetch stage, so in this example, the fetch stage represents not only the time the instruction spends in the IQ but also the time required for the instruction to be loaded from system memory, beginning in clock cycle 3.

During clock cycle 2, the target instruction for the **br** instruction is not in the instruction cache; therefore, a memory access must occur. During clock cycle 5, the address of the block of instructions is sent to the system bus. During clock cycle 9, two instructions (64 bits) are returned from memory on the first beat and are forwarded both to the cache and instruction fetcher.

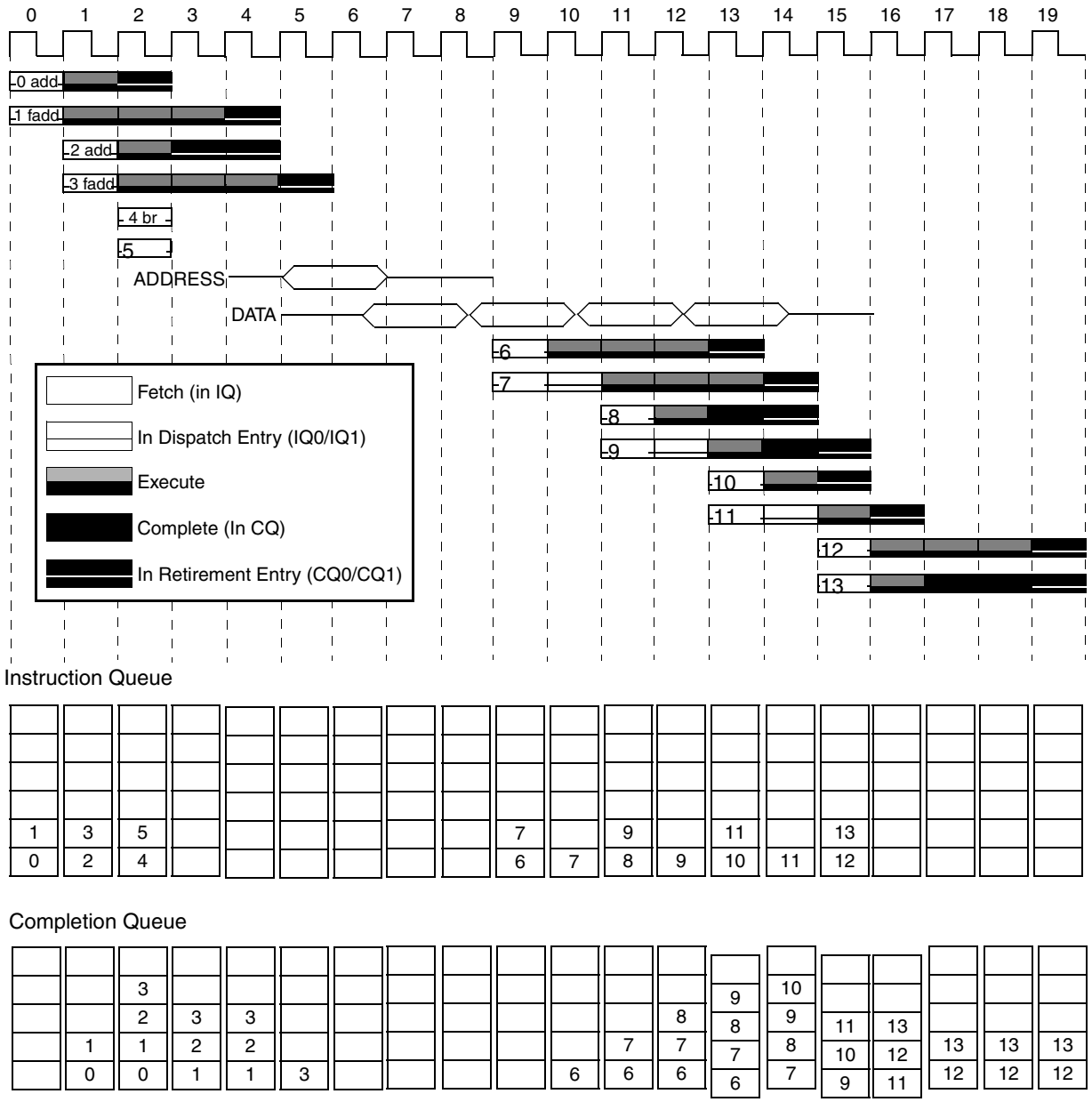


Figure 7-7. Instruction Timing—Cache Miss

7.3.3 Instruction Dispatch and Completion Considerations

Several factors affect the ability of the core to dispatch instructions at a peak rate of two per cycle—the availability of the execution unit, destination rename registers, and completion queue, as well as the handling of completion-serialized instructions. Several of these limiting factors are illustrated in the previous instruction timing examples.

To reduce dispatch unit stalls due to instruction data dependencies, the core provides a single-entry reservation station for the FPU, SRU, and each IU, and a two-entry reservation station for the LSU. If a data dependency keeps an instruction from starting execution, that instruction is dispatched to the reservation station associated with its execution unit (and the rename registers are assigned), thereby freeing the positions in the instruction queue so instructions can be dispatched to other execution units. Execution begins during the same clock cycle that the rename buffer is updated with the data the instruction is dependent on.

If both instructions in IQ0 and IQ1 require the same execution unit, the instruction in IQ1 cannot be dispatched until the first instruction proceeds through the pipeline and provides the subsequent instruction with a vacancy in the requested execution unit.

The completion unit maintains program order after instructions are dispatched, guaranteeing in-order completion and a precise interrupt model. Completing an instruction committing execution results to the architected destination registers. In-order completion ensures the correct architectural state when the core must recover from a mispredicted branch or an interrupt.

The core can execute instructions out-of-order, but in-order completion by the completion unit ensures a precise interrupt mechanism. Program-related interrupts are signaled when the instruction causing the interrupt reaches the last position in the completion queue. Prior instructions are allowed to complete before the interrupt is taken.

7.3.3.1 Rename Register Operation

To avoid contention for a given register file location, the core provides rename registers for holding instruction results before the completion commits them to the architected register. There are five GPR rename registers, four FPR rename registers, and one each for the CR, LR, and CTR.

When an instruction dispatches to its execution unit, any required rename registers are allocated for the results of that instruction. If an instruction is dispatched to the reservation station associated with an execution unit due to a data dependency, the dispatcher also provides a tag to the execution unit identifying the rename register that forwards the required data at completion. When the source data reaches the rename register, execution can begin.

Instruction results are transferred from rename registers to architected registers when an instruction is retired from the CQ after any associated interrupts are handled and any predicted branch conditions preceding it in the CQ are resolved. If a branch prediction is incorrect, the instructions following the branch are flushed from the CQ and any results of those instructions are flushed from the rename registers.

7.3.3.2 Instruction Serialization

Although the core can dispatch and complete two instructions per cycle, serializing instructions can be used to limit dispatch and completion to one instruction per cycle. Serialization falls into three categories—completion, dispatch, and refetch serialization, which are described as follows:

- Completion serialized instructions are held in the execution unit until all prior instructions in the completion unit have been retired. Completion serialization is used for instructions that access or modify a resource for which no rename register exists. Results from these instructions are not available or forwarded for subsequent instructions until the serializing instruction is retired. Instructions that are completion serialized are as follows:
  - Instructions (with the interrupt of integer add and compare instructions) executed by the system register unit (SRU)
  - Floating-point instructions that access or modify the FPSCR or CR (**mtfsb1**, **mcrfs**, **mtfsfi**, **mffs**, and **mtfsf**).
  - Instructions that manage caches and TLBs
  - Instructions that directly access the GPRs (load and store multiple word and load and store string instructions)
  - Instructions defined by the architecture to have synchronizing behavior
- Dispatch serialized inhibit the dispatching of subsequent instructions until the serializing instruction is retired. Dispatch serialization is used for instructions that access renamed resources used by the dispatcher, and for instructions requiring refetch serialization, including the following:
  - The load multiple instructions, **lmw**, **lswi**, and **lswx**.
  - The **mtspr**(XER) and **mcrxr** instructions
  - The synchronizing instructions, **sync**, **isync**, **mtmsr**, **rfi**, **rfci** and **sc**.
- Refetch serialized instructions inhibit dispatching of subsequent instructions and force the refetching of subsequent instructions after the serializing instructions are retired. The context synchronizing instruction, **isync**, is refetch serializing.

7.3.3.3 Execution Unit Considerations

As previously noted, the e300 core can dispatch and retire two instructions per clock cycle. The peak dispatch rate is affected by the availability of execution units on each clock cycle.

For an instruction to be dispatched, the required execution unit must be available. The dispatcher monitors the availability of all execution units and suspends instruction dispatch if the required execution unit is unavailable. An execution unit may not be available if it can accept and execute only one instruction per cycle or if an execution unit's pipeline becomes full, which may occur if instruction execution takes more clock cycles than the number of pipeline stages in the unit and additional instructions are dispatched to that unit to fill the remaining pipeline stages.

7.4 Execution Unit Timings

The following sections describe instruction timing considerations for each execution unit.

7.4.1 Branch Processing Unit Execution Timing

Flow control operations (conditional branches, unconditional branches, and traps) are typically expensive to execute in most machines because they disrupt normal flow in the instruction stream. When a change in program flow occurs, the IQ must be reloaded with the target instruction stream. During this time the execution units will be idle. However, previously dispatched instructions will continue to execute while the new instruction stream makes its way into the IQ.

Performance features such as branch folding and static branch prediction help minimize penalties associated with flow control operations. The timing for branch instruction execution is determined by many factors including the following:

- Whether the branch requires prediction
- Whether the branch is predicted as taken or not taken
- Whether the branch is taken
- Whether the target instruction stream is in the on-chip cache
- Whether the prediction is correct

7.4.1.1 Branch Folding

When a branch instruction is encountered by the fetcher, the BPU immediately tries to pull that instruction out of the instruction stream and resolve it. When the BPU removes the branch instruction from the stream, the subsequent instruction is shifted down to take the place of the removed branch instruction. This technique is called branch folding. Often, it eliminates the penalties of flow control instructions because instruction execution proceeds as though the branch were never there.

If the folded branch instruction changes program flow (the branch is said to be taken), the BPU immediately requests the instructions at the new target from the on-chip cache. In most cases, the new instructions arrive in the IQ before any bubbles are introduced into the execution units. If the folded branch does not change program flow (the branch is not taken), the branch instruction is already removed and execution continues as if there were never a branch in the original sequence.

When a conditional branch cannot be resolved due to a CR data dependency, the branch is executed by means of static branch prediction and instruction fetching proceeds down the predicted path. If the prediction is incorrect when the branch is resolved, the IQ and all subsequently executed instructions are purged, instructions executed before the predicted branch are allowed to complete, and instruction fetching resumes down the correct path.

There are several situations where instruction sequences create dependencies that prevent a branch instruction from being resolved immediately, thereby causing execution of the subsequent instruction stream based on the predicted outcome of the branch instruction. The instruction sequences, and the resulting action of the branch instruction is described as follows:

- An **mtspr**(LR) followed by a **bclr**—Fetching is stopped and the branch waits for the **mtspr** to execute.
- An **mtspr**(CTR) followed by a **bcctr**—Fetching is stopped and the branch waits for the **mtspr** to execute.

- An **mtspr**(CTR) followed by a **bc**(CTR)—Fetching is stopped and the branch waits for the **mtspr** to execute. (Note: Branch conditions can be a function of the CTR and CR; if the CTR condition is sufficient to resolve the branch, then a CR-dependency is ignored.)
- A **bc**(CTR) followed by another **bc**(CTR)—Fetching is stopped, and the second branch waits for the first to be completed.
- A **bc**(CTR) followed by a **bcctr**—Fetching is stopped, and the **bcctr** waits for the first branch to be completed.
- A branch(LK = 1) followed by a branch(LK = 1)—Fetching is stopped, and the second branch waits for the first branch to be completed. (Note: a **bl** instruction does not have to wait for a branch(LK = 1) to complete.)
- A **bc**(based-on-CR) waiting for resolution due to a CR-dependency followed by a **bc**(based-on-CR)—Fetching is stopped and the second branch waits for the first CR-dependency to be resolved.

7.4.1.2 Static Branch Prediction

Static branch prediction allows software (for example, compilers) to give a hint to the machine hardware about the direction the branch is likely to take. When a branch instruction encounters a data dependency, the BPU waits for the required condition code to become available. Rather than stalling instruction dispatch until the source operand is ready, the core predicts the likely path and instructions are fetched and executed along that path. When the branch operand becomes available, the branch is evaluated. If the prediction is correct, program flow continues along that path uninterrupted; otherwise, the processor backs up and program flow resumes along the correct path.

If the target address of the branch (link or count register) is modified by an instruction that appears before the branch instruction, the BPU waits until the target address is available.

The core executes through one level of prediction. The processor may not predict a branch if a prior branch instruction is still unresolved.

The number of instructions that can be executed after branch prediction is limited by the fact that instructions in the predicted stream cannot update the register files or memory until the branch is resolved. That is, instructions may be dispatched and executed, but cannot reach the write-back stage in the completion unit, instead, it stalls in the completion queue. When CQ is full, no more instructions can be dispatched.

In the case of a misprediction, the core is able to redirect the machine state rather effortlessly because the programing model has not been updated. When a branch is found to be mispredicted, all instructions that were dispatched subsequent to the predicted branch instruction are simply flushed from the completion queue, and their results flushed from the rename registers. No architected register state needs to be restored because no architected register state was modified by the instructions following the unresolved predicted branch.

7.4.1.2.1 Predicted Branch Timing Examples

Figure 7-8 shows how both taken and non-taken branches are handled and how the e300 core handles both correct and incorrect predictions. The example shows the timing for the following instruction sequence (note that the first **bc** instruction is correctly taken, whereas the second **bc** is incorrectly predicted):

```
   0  add
   1  add
   2  bc
   3  mulhw
   4  bc T0
   5  fadd
   6  and
T0      add
T1      add
T2      add
T3      add
T4      and
T5      or
```

0. During clock cycle 0, instructions 0 and 1 are dispatched in the beginning of clock cycle 1.
1. In clock cycle 1, instructions 2 and 3 are fetched in the IQ. Instruction 2 is a branch instruction that updates the CTR and instruction 3 is a **mulhw** instruction on which instruction 4 depends. Instruction 0 enters the IU. Instruction 1 has a single-cycle stall.
2. In clock cycle 2, instructions 4 (a second **bc** instruction) and 5 are fetched. The second **bc** instruction is predicted as taken. It can be folded, but it cannot be resolved until instruction 3 writes back. Instruction 0 completes at the end of this cycle. Instruction 1 is dispatched to the IU. Instruction 2 takes entry in the CQ.
3. In clock cycle 3, target instruction T0 and T1 are fetched. Instructions 1 and 2 complete, instruction 4 has been folded, and instruction 5 has been flushed from the IQ. Instruction 3 is assigned to CQ2.
4. In clock cycle 4, target instructions T2 and T3 are fetched. IU instructions T0 and T1 have multiple stalls as one execution possible in a clock cycle. Instruction 3 is assigned to CQ0.
5. In clock cycle 5, instruction 3, on which the second branch instruction depended, writes back and the branch prediction is proven incorrect. Even though T0 is in CQ0, where it could be written back, it is not because the prediction was incorrect. All target instructions are flushed from their positions in the pipeline at the end of this clock cycle, as there are many results in the rename registers.

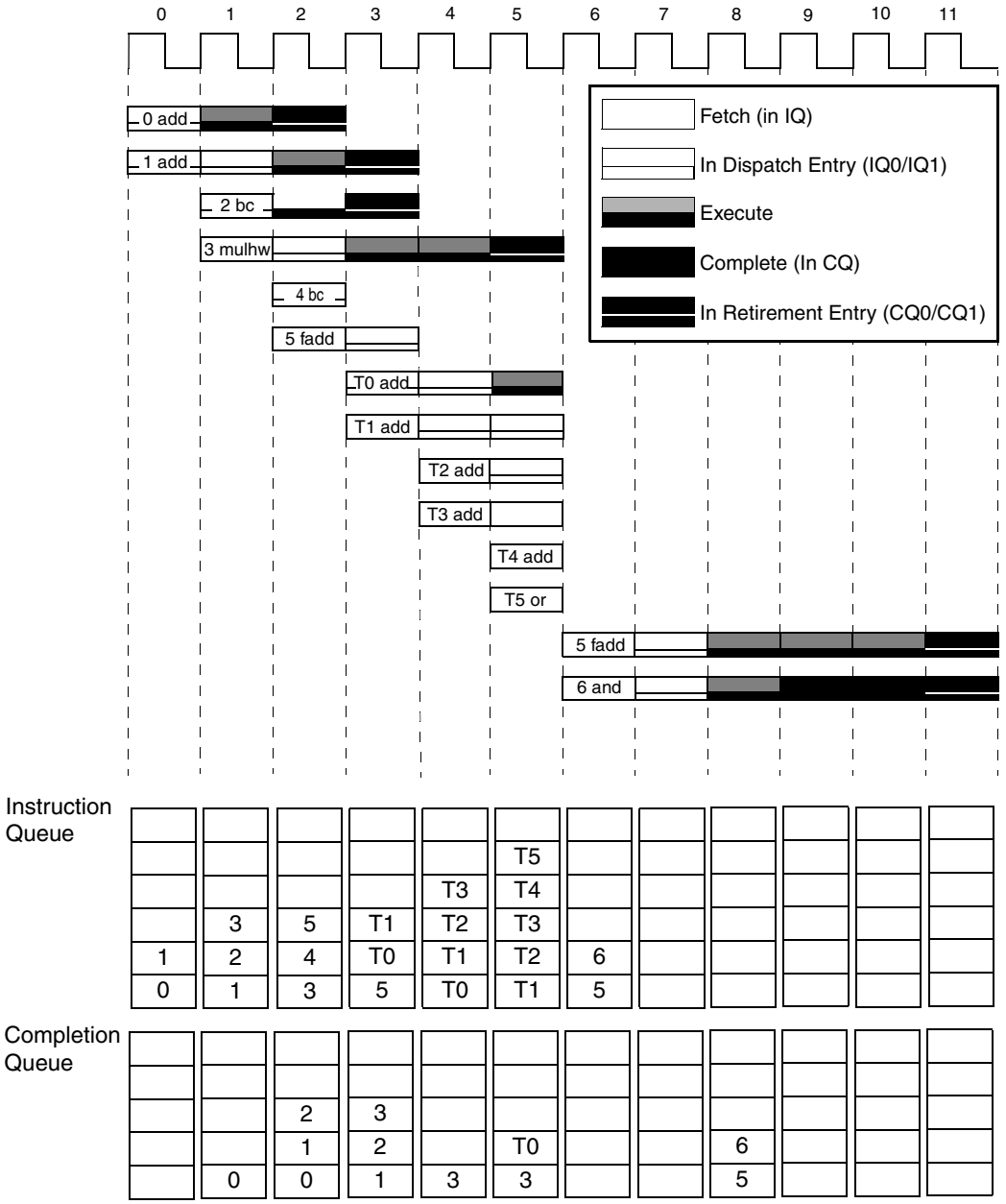


Figure 7-8. Branch Instruction Timing

After one clock cycle required to refetch the original instruction stream, instruction 5, the same instruction that was fetched in clock cycle 2, is brought back into the IQ from the instruction cache, along with one other.

7.4.2 Integer Unit Execution Timing

The integer unit executes all integer and bit-field computational instructions. Many of these instructions execute in a single clock cycle. The integer unit has one execute stage so when a multiple-cycle integer instruction is executed, no other integer instructions can also begin to execute. In the e300c2, e300c3,

however, each of the two execution units can execute one multiply for a total of two multiply instructions executed in parallel. See Table 7-4 for integer instruction execution timing.

Figure 7-9 shows how the e300c1 core handles integer instructions. Execution of multiply half-word unsigned instructions may take 2–6 cycles in the e300c1. The example shows the worst-case timing for the following instruction sequence:

```
0  mulhwu
1  add
2  mulhwu
3  add
```

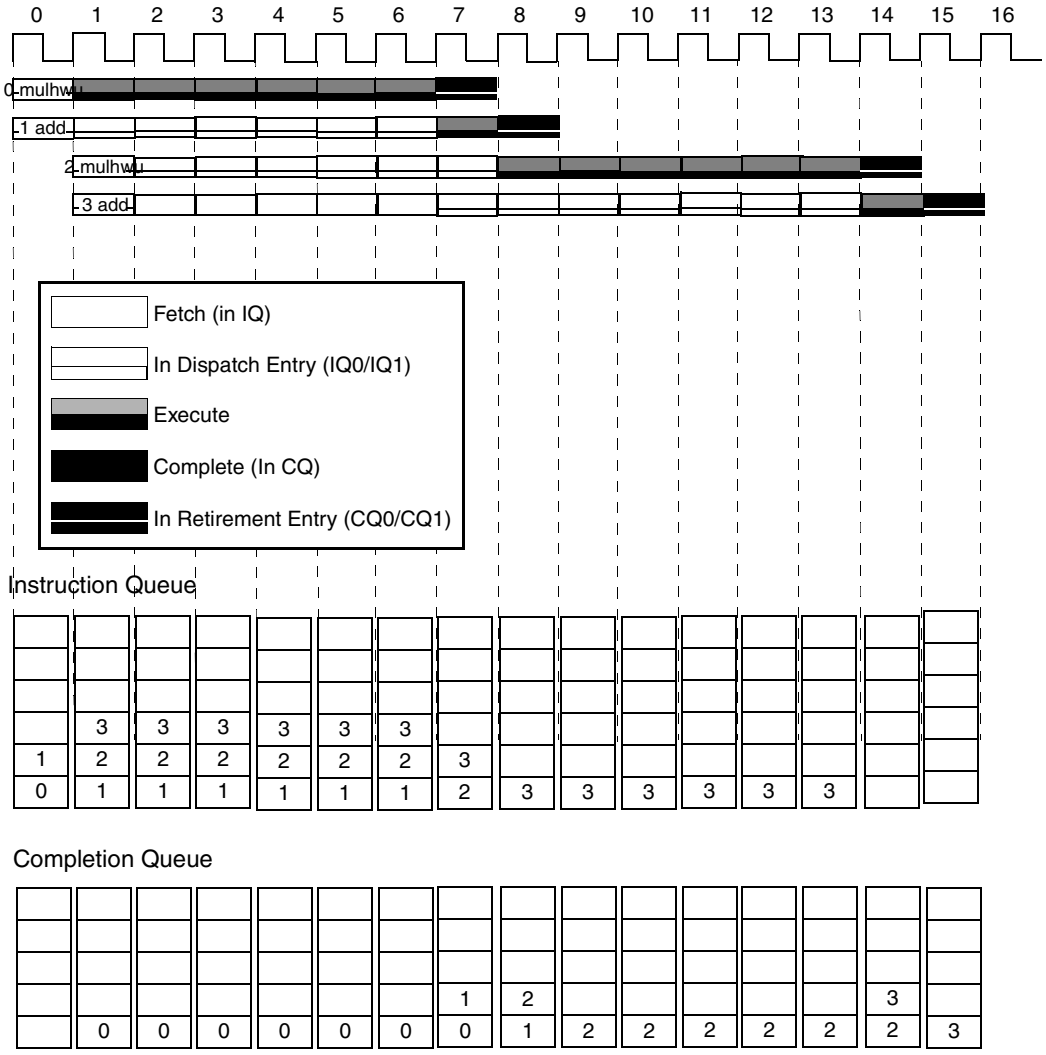


Figure 7-9. Instruction Timing—Integer Execution in the e300c1Core

Figure 7-10 shows how the e300c2, e300c3 handle integer instructions. The example shows the timing for the same instruction sequence as the previous example. The two integer units in the e300c2, e300c3, along

with the faster multipliers, result in higher throughput of integer instructions when compared to the e300c1.

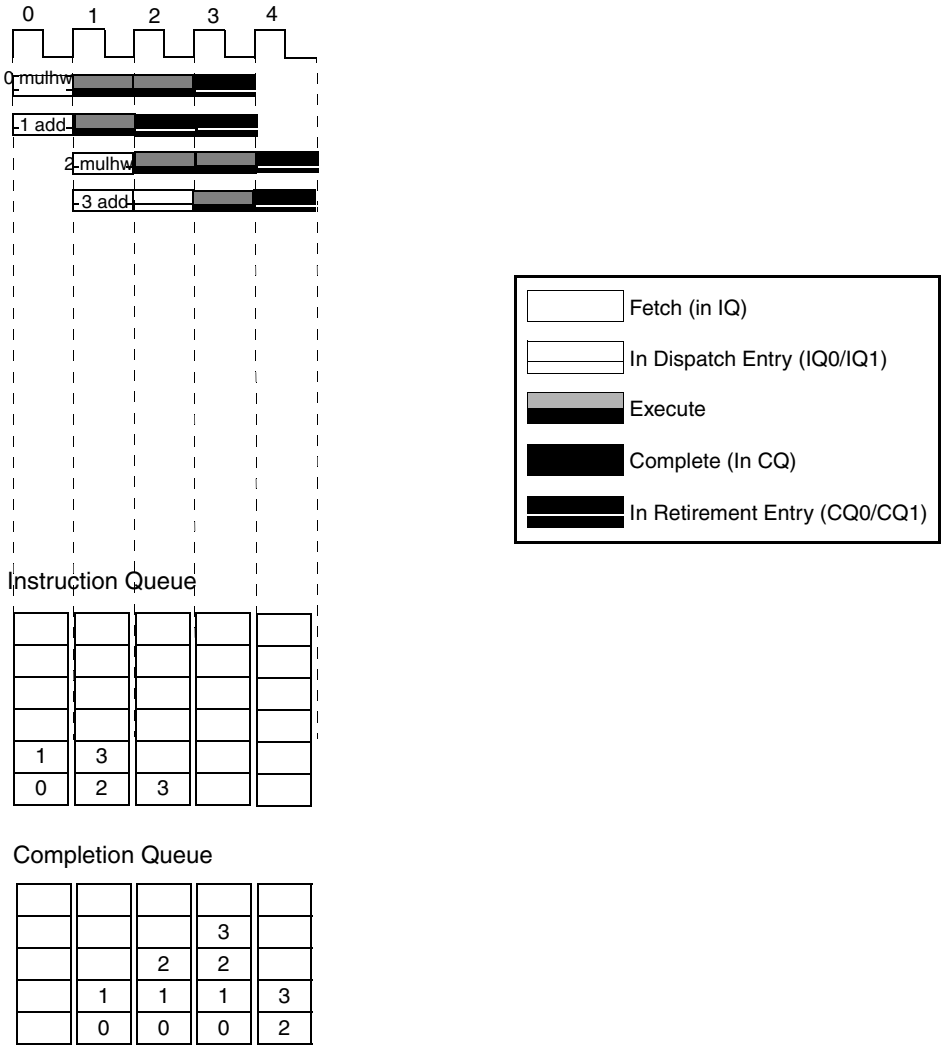


Figure 7-10. Instruction Timing—Integer Execution in the e300c2 and e300c3

7.4.3 Floating-Point Unit Execution Timing

Floating-point instructions are not supported on the e300c2 core. This section is for comparison purposes only with the e300 core. The FPU on the e300 core executes all floating-point computational instructions. The LSU performs integer floating-point loads and stores. Execution of most floating-point instructions is pipelined within the FPU, allowing up to three instructions to be executing in the FPU concurrently. On the e300 core, the FPU is pipelined so a single-precision multiply-add instruction can be issued and completed every clock cycle. While most floating-point instructions execute with three- or four-cycle latency, and one- or two-cycle throughput, three instructions (**fdivs**, **fdiv**, and **fres**) execute with latencies of 18 to 33 cycles. The **fdivs**, **fdiv**, **fres**, **mtfsb0**, **mtfsb1**, **mtfsfi**, **mffs**, and **mtfsf** instructions block the floating-point unit pipeline until they complete execution, and thereby inhibit the dispatch of additional floating-point instructions. Except for the **mcrfs** instruction, all floating-point instructions immediately

forward their CR results to the BPU for fast branch resolution without waiting for the instruction to be retired by the completion unit, and the CR updated. See Table 7-5 for floating-point instruction execution timing.

7.4.4 Load/Store Unit Execution Timing

The LSU executes all floating-point and integer loads and stores. It also executes other instructions that address memory. The execution of most load and store instructions is pipelined. The LSU has two pipeline stages; the first is for effective address calculation and MMU translation, and the second is for accessing the physically addressed memory. Load and store instructions have a two-cycle latency and one-cycle throughput. Floating-point loads or stores are not supported on the e300c2 core.

If operands are misaligned, additional latency may be required either for an alignment interrupt to be taken or for additional bus accesses. Load instructions that miss in the cache prevent subsequent cache accesses during the cache line refill. See Table 7-6 for load and store instruction execution timing.

7.4.5 System Register Unit Execution Timing

Most SRU instructions access or modify nonrenamed registers, or directly access renamed registers. They generally execute in a serial manner. Results from these instructions are not available or forwarded for use by subsequent instructions until the instruction completes and is retired. The SRU can also execute the integer instructions **addi**, **addis**, **add**, **addo**, **cmpi**, **cmp**, **cmpli**, and **cmpl** without serialization and in parallel with another integer instruction. Refer to Section 7.3.3.2, “Instruction Serialization,” for additional information on serializing instructions and Table 7-2, Table 7-3, and Table 7-4 for SRU instruction execution timing.

7.5 Memory Performance Considerations

Due to the e300 core instruction throughput of three instructions per clock cycle, lack of data bandwidth can become a performance bottleneck. For the core to approach its potential performance levels, it must be able to read and write data quickly and efficiently. If there are many processors in a system environment, one processor may experience long memory latencies while another bus master (for example, a direct-memory access controller) is using the external bus.

To alleviate this possible contention, the e300 core provides three memory update modes—copy-back, write-through, and cache-inhibit. Each page of memory is specified to be in one of these modes. If a page is in copy-back mode, data being stored to that page is written only to the on-chip cache. If a page is in write-through mode, writes to that page update the on-chip cache on hits and always update main memory. If a page is cache-inhibited, data in that page will never be stored in the on-chip cache. All three of these modes of operation have advantages and disadvantages. A decision as to which mode to use depends on the system environment as well as the application.

The following sections describe how performance is impacted by each memory update mode. For details about the operation of the on-chip cache and the memory update modes, see Chapter 4, “Instruction and Data Cache Operation.”

7.5.1 Copy-Back Mode

When data is stored in a location marked as copy back, store operations for cacheable data do not necessarily cause an external bus cycle to update memory. Instead, memory updates only occur on modified line replacements, cache flushes, or when another processor attempts to access a specific address for which there is a corresponding modified cache entry. For this reason, copy-back mode may be preferred when external bus bandwidth is a potential bottleneck—for example, in a multiprocessor environment. Copy-back mode is also well suited for data that is closely coupled to a processor, such as local variables.

If more than one device uses data stored in a page marked as copy back, snooping must be enabled to allow copy-back operations and cache invalidations of modified data. The e300 core implements snooping hardware to prevent other devices from accessing invalid data. When bus snooping is enabled, depending on the device integration, the processor can monitor the transactions of the other devices. For example, if another device accesses a memory location and its memory-coherent (M) bit is set and the core on-chip cache has a modified value for that address, the processor preempts the bus transaction and updates memory with the cache data. If the cache contents associated with the snooped address are unmodified, the core invalidates the cache block. The other device can then attempt an access to the updated address. See Chapter 4, “Instruction and Data Cache Operation.”

Copy-back mode provides complete cache/memory coherency as well as maximizing available external bus bandwidth.

7.5.2 Write-Through Mode

Store operations to memory in write-through mode always update memory as well as the on-chip cache (on cache hits). Write-through mode is used when the data in the cache must always agree with external memory (for example, video memory), when shared (global) data may be used frequently, or when allocation of a cache line on a cache miss is undesirable. Automatic copy back of cached data is not performed if that data is from a memory page marked as write-through mode because valid cache data always agrees with memory.

Stores to memory that are in write-through mode may cause a decrease in performance. Each time a store is performed to memory in write-through mode, the bus is potentially busy for the extra clock cycles required to update memory; therefore, load operations that miss the on-chip cache must wait while the external store operation completes.

7.5.3 Cache-Inhibited Accesses

Data for a page marked cache-inhibited cannot be stored in the on-chip cache.

Areas of the memory map can be cache-inhibited by the operating system. If a cache-inhibited access hits in the on-chip cache, the corresponding cache line is invalidated. If the line is marked modified, it is copied back to memory before being invalidated.

In summary, the copy-back mode allows both load and store operations to use the on-chip cache. The write-through mode allows load operations to use the on-chip cache, but store operations cause a memory access and a cache update if the data is already in the cache. Lastly, the cache-inhibited mode causes memory access for both loads and stores.

7.6 Instruction Scheduling Guidelines

The performance of the core can be improved by avoiding resource conflicts and promoting parallel utilization of execution units through efficient instruction scheduling. Instruction scheduling on the core can be improved by observing the following guidelines:

- Implement good static branch prediction (setting of y bit in BO field).
- When branch prediction is uncertain, or an even probability, predict fall through.
- To reduce mispredictions, separate the instruction that sets CR bits from the branch instruction that evaluates them; separation by more than nine instructions ensures that the CR bits will be immediately available for evaluation.
- When branching conditionally to a location specified by count registers (CTRs) or link registers (LRs), or when branching conditionally based on the value in the count register, separate the **mtspr** instruction that initializes the CTR or LR from the branch instruction performing the evaluation. Separation of the branch and **mtspr** instruction by more than nine instructions ensures the register values will be immediately available for use by the branch instruction.
- Schedule instructions such that they can dual dispatch.
- Schedule instructions to minimize stalls when an execution unit is busy.
- Avoid using serializing instructions.
- Schedule instructions to avoid dispatch stalls due to renamed resource limitations.
  - — Only five instructions can be in execute-complete stage at any one time.
  - — Only five GPR destinations can be in execute-complete-deallocate stage at any one time. Note that load with update address instructions use two destination registers.
  - — Only four FPR destinations can be in execute-complete-deallocate stage at any one time.

7.6.1 Branch, Dispatch, and Completion Unit Resource Requirements

This section describes the specific resources required to avoid stalls during branch resolution, instruction dispatching, and instruction completion.

7.6.1.1 Branch Resolution Resource Requirements

The following is a list of branch instructions and the resources required to avoid stalling the fetch unit in the course of branch resolution:

- The **bclr** instruction requires LR availability.
- The **bcctr** instruction requires CTR availability.
- Branch and link instructions require shadow LR availability.
- The branch conditional on counter decrement and CR condition requires CTR availability or the CR condition must be false, and the core cannot be executing instructions following an unresolved predicted branch when the branch is encountered by the BPU.
- The branch conditional on CR condition cannot be executed following an unresolved predicted branch instruction.

7.6.1.2 Dispatch Unit Resource Requirements

The following is a list of resources required to avoid stalls in the dispatch unit. Note that the two dispatch buffers, IQ0 and IQ1, are at the bottom of the instruction queue:

- Requirements for dispatching from IQ0 are as follows:
  - Needed execution unit available
  - Needed GPR rename registers available
  - Needed FPR rename registers available
  - Completion queue is not full
  - Instruction is dispatch serialized and completion buffer is empty
  - A dispatch serialized instruction is not currently being executed
- Requirements for dispatching from IQ1 are as follows:
  - Instruction in IQ0 must dispatch
  - Instruction dispatched by IQ0 is not dispatch serialized
  - Needed execution unit is available (after dispatch from IQ0)
  - Needed GPR rename registers are available (after dispatch from IQ0)
  - Needed FPR rename register is available (after dispatch from IQ0)
  - Completion queue is not full (after dispatch from IQ0)
  - Instruction dispatched from IQ1 is not dispatch serialized

7.6.1.3 Completion Unit Resource Requirements

The following is a list of resources required to avoid stalls in the completion unit; note that the two completion buffers are described as CQ0 and CQ1, where CQ0 is the entry at the end of the completion queue:

- Requirements for completing an instruction from CQ0 are as follows:
  - Instruction in CQ0 must be finished
  - Instruction in CQ0 must not follow an unresolved predicted branch
  - Instruction in CQ0 must not cause an interrupt
- Requirements for completing an instruction from CQ1 are as follows:
  - Instruction in CQ0 must complete in same cycle
  - Instruction in CQ1 must be finished
  - Instruction in CQ1 must not follow an unresolved predicted branch
  - Instruction in CQ1 must not cause an interrupt
  - Instruction in CQ1 must be an integer or load instruction
  - Number of CR updates from both CQ0 and CQ1 must not exceed one
  - Number of GPR updates from both CQ0 and CQ1 must not exceed two
  - Number of FPR updates from both CQ0 and CQ1 must not exceed one

7.7 Instruction Latency Summary

Table 7-1 through Table 7-6 list the latencies associated with each instruction executed by the e300 core. Note that the instruction latency tables contain no 64-bit architected instructions. These instructions will trap to an illegal instruction interrupt handler when encountered. Recall that the term latency is defined as the total time it takes to execute an instruction and make ready the results of that instruction.

Table 7-1 provides the latencies for the branch instructions.

Table 7-1. Branch Instructions

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles)[1]
bc[**l**][**a**]	16	—	BPU	1
b[**l**][**a**]	18	—	BPU	1
bclr[**l**]	19	016	BPU	1
bcctr[**l**]	19	528	BPU	1

[1] These operations may be folded for an effective cycle time of 0.

Table 7-2 provides the latencies for the system register instructions.

Table 7-2. System Register Instructions

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles)
sc	17	- -1	SRU	3
rfi	19	050	SRU	3
rfci	19	051	SRU	3
isync	19	150	SRU	1&
mfmsr	31	083	SRU	1
mtmsr	31	146	SRU	2
mtsr	31	210	SRU	2
mtsrin	31	242	SRU	2
mfspr (not I/DBATs)	31	339	SRU	1
mfspr (DBATs)	31	339	SRU	3&
mfspr (IBATs)	31	339	SRU	3&
mtspr (not IBATs)	31	467	SRU	2 (XER-&)
mtspr (IBATs)	31	467	SRU	2&
mfsr	31	595	SRU	3&
sync	31	598	SRU	1&
mfsrin	31	659	SRU	3&
eieio	31	854	SRU	1
mftb	31	371	SRU	1

Table 7-2. System Register Instructions (continued)

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles)
mttb	31	467	SRU	1

Note: Cycle times marked with & require a variable number of cycles due to serialization.

Table 7-3 provides the latencies for the condition register logical instructions.

Table 7-3. Condition Register Logical Instructions

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles)
mcrf	19	000	SRU	1
crnor	19	033	SRU	1
crandc	19	129	SRU	1
crxor	19	193	SRU	1
crnand	19	225	SRU	1
crand	19	257	SRU	1
creqv	19	289	SRU	1
crorc	19	417	SRU	1
cror	19	449	SRU	1
mfcr	31	019	SRU	1
mtcrf	31	144	SRU	1
mcrxr	31	512	SRU	1&

Note: Cycle times marked with & require a variable number of cycles due to serialization.

Table 7-4 provides the latencies for the integer instructions.

Table 7-4. Integer Instructions

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles) in e300c1	Latency (in Cycles) in e300c2, e3000c3
twi	03	—	Integer	2	2
mulli	07	—	Integer	2,3	2
subfic	08	—	Integer	1	1
cmpli	10	—	Integer & SRU	1^	1^
cmpi	11	—	Integer & SRU	1^	1^
addic	12	—	Integer	1	1
addic.	13	—	Integer	1	1

Table 7-4. Integer Instructions (continued)

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles) in e300c1	Latency (in Cycles) in e300c2, e3000c3
addi	14	—	Integer & SRU	1	1
addis	15	—	Integer & SRU	1	1
rlwimi[.]	20	—	Integer	1	1
rlwinm[.]	21	—	Integer	1	1
rlwnm[.]	23	—	Integer	1	1
ori	24	—	Integer	1	1
oris	25	—	Integer	1	1
xori	26	—	Integer	1	1
xoris	27	—	Integer	1	1
andi.	28	—	Integer	1	1
andis.	29	—	Integer	1	1
cmp	31	000	Integer & SRU	1^	1^
tw	31	004	Integer	2	2
subfc[**o**][.]	31	008	Integer	1	1
addc[**o**][.]	31	010	Integer	1	1
mulhwu[.]	31	011	Integer	2,3,4,5,6	2
slw[.]	31	024	Integer	1	1
cntlzw[.]	31	026	Integer	1	1
and[.]	31	028	Integer	1	1
cmpl	31	032	Integer & SRU	1^	1^
subf[.]	31	040	Integer	1	1
andc[.]	31	060	Integer	1	1
mulhw[.]	31	075	Integer	2,3,4,5	2
neg[**o**][.]	31	104	Integer	1	1
nor[.]	31	124	Integer	1	1
subfe[**o**][.]	31	136	Integer	1	1
adde[**o**][.]	31	138	Integer	1	1
subfze[**o**][.]	31	200	Integer	1	1
addze[**o**][.]	31	202	Integer	1	1
subfme[**o**][.]	31	232	Integer	1	1
addme[**o**][.]	31	234	Integer	1	1
mull[**o**][.]	31	235	Integer	2,3,4,5	2
add[**o**][.]	31	266	Integer & SRU[1]	1	1

Table 7-4. Integer Instructions (continued)

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles) in e300c1	Latency (in Cycles) in e300c2, e3000c3
eqv[.]	31	284	Integer	1	1
xor[.]	31	316	Integer	1	1
orc[.]	31	412	Integer	1	1
or[.]	31	444	Integer	1	1
divwu[**o**][.]	31	459	Integer	20	20
nand[.]	31	476	Integer	1	1
divw[**o**][.]	31	491	Integer	20	20
srw[.]	31	536	Integer	1	1
sraw[.]	31	792	Integer	1	1
srawi[.]	31	824	Integer	1	1
extsh[.]	31	922	Integer	1	1
extsb[.]	31	954	Integer	1	1

Note: ^ indicates that the cycle time immediately forwards their CR results to the BPU for fast branch resolution.

[1] The SRU can only execute the **add** and **add**[**o**] instructions.

Table 7-5 provides the latencies for the floating-point instructions.

Table 7-5. Floating-Point Instructions

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles)
fdivs[.]	59	018	FPU	18^
fsubs[.]	59	020	FPU	1-1-1^
fadds[.]	59	021	FPU	1-1-1^
fres[.]	59	024	FPU	18^
fmuls[.]	59	025	FPU	1-1-1^
fmsubs[.]	59	028	FPU	1-1-1^
fmadds[.]	59	029	FPU	1-1-1^
fnmsubs[.]	59	030	FPU	1-1-1^
fnmadds[.]	59	031	FPU	1-1-1^
fcmpu	63	000	FPU	1-1-1^
frsp[.]	63	012	FPU	1-1-1^
fctiw[.]	63	014	FPU	1-1-1^
fctiwz[.]	63	015	FPU	1-1-1^

Table 7-5. Floating-Point Instructions (continued)

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles)
fdiv[.]	63	018	FPU	33^
fsub[.]	63	020	FPU	1-1-1^
fadd[.]	63	021	FPU	1-1-1^
fsel[.]	63	023	FPU	1-1-1^
fmul[.]	63	025	FPU	2-1-1^
frsqrte[.]	63	026	FPU	1-1-1^
fmsub[.]	63	028	FPU	2-1-1^
fmadd[.]	63	029	FPU	2-1-1^
fnmsub[.]	63	030	FPU	2-1-1^
fnmadd[.]	63	031	FPU	2-1-1^
fcmpo	63	032	FPU	1-1-1^
mtfsb1[.]	63	038	FPU	1-1-1&^
fneg[.]	63	040	FPU	1-1-1^
mcrfs	63	064	FPU	1-1-1&
mtfsb0[.]	63	070	FPU	1-1-1&^
fmr[.]	63	072	FPU	1-1-1^
mtfsfi[.]	63	134	FPU	1-1-1&^
fnabs[.]	63	136	FPU	1-1-1^
fabs[.]	63	264	FPU	1-1-1^
mffs[.]	63	583	FPU	1-1-1&^
mtfsf[.]	63	711	FPU	1-1-1&^

Note: Cycle times marked with & require a variable number of cycles due to completion serialization.
Cycle times marked with ^ immediately forward their CR results to the BPU for fast branch resolution.
Cycle times marked with a - specify the number of clock cycles in each pipeline stage. Instructions with a single entry in the cycles column are not pipelined.

NOTE

Floating point instructions are not supported on the e300c2.

Table 7-6 provides latencies for the load and store instructions.

Table 7-6. Load and Store Instructions

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles)
lwarx	31	020	LSU	2:1
icbt	31	022	LSU	2

Table 7-6. Load and Store Instructions (continued)

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles)
lwzx	31	023	LSU	2:1
dcbst	31	054	LSU	2/5&
lwzux	31	055	LSU	2:1
dcbf	31	086	LSU	2/5&
lbzx	31	087	LSU	2:1
lbzux	31	119	LSU	2:1
stwcx.	31	150	LSU	8
stwx	31	151	LSU	2:1
stwux	31	183	LSU	2:1
stbx	31	215	LSU	2:1
dcbtst	31	246	LSU	2
stbux	31	247	LSU	2:1
dcbt	31	278	LSU	2
lhzx	31	279	LSU	2:1
tlbie	31	306	LSU	3&
lhzux	31	311	LSU	2:1
lhax	31	343	LSU	2:1
lhaux	31	375	LSU	2:1
sthx	31	407	LSU	2:1
sthux	31	439	LSU	2:1
dcbi	31	470	LSU	2&
lswx	31	533	LSU	2 + *n*&
lwbrx	31	534	LSU	2:1
lfsx	31	535	LSU	2:1
tlbsync	31	566	LSU	2&
lfsux	31	567	LSU	2:1
lswi	31	597	LSU	2 + *n*&
lfdx	31	599	LSU	2:1
lfdux	31	631	LSU	2:1
stswx	31	661	LSU	1 + *n*&
stwbrx	31	662	LSU	2:1
stfsx	31	663	LSU	2:1
stfsux	31	695	LSU	2:1
stswi	31	725	LSU	1 + *n*&

Table 7-6. Load and Store Instructions (continued)

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles)
stfdx	31	727	LSU	2:1
stfdux	31	759	LSU	2:1
lhbrx	31	790	LSU	2:1
sthbrx	31	918	LSU	2:1
tlbld	31	978	LSU	2&
icbi	31	982	LSU	3&
stfiwx	31	983	LSU	2:1
tlbli	31	1010	LSU	3&
dcbz	31	1014	LSU	10&
lwz	32	—	LSU	2:1
lwzu	33	—	LSU	2:1
lbz	34	—	LSU	2:1
lbzu	35	—	LSU	2:1
stw	36	—	LSU	2:1
stwu	37	—	LSU	2:1
stb	38	—	LSU	2:1
stbu	39	—	LSU	2:1
lhz	40	—	LSU	2:1
lhzu	41	—	LSU	2:1
lha	42	—	LSU	2:1
lhau	43	—	LSU	2:1
sth	44	—	LSU	2:1
sthu	45	—	LSU	2:1
lmw	46	—	LSU	2 + *n*&
stmw	47	—	LSU	1 + *n*&
lfs	48	—	LSU	2:1
lfsu	49	—	LSU	2:1
lfd	50	—	LSU	2:1
lfdu	51	—	LSU	2:1
stfs	52	—	LSU	2:1
stfsu	53	—	LSU	2:1
stfd	54	—	LSU	2:1

Table 7-6. Load and Store Instructions (continued)

Mnemonic	Primary Opcode	Extended Opcode	Unit	Latency (in Cycles)
stfdu	55	—	LSU	2:1

Note: Cycle times marked with & require a variable number of cycles due to serialization.
Cycle times marked with a / specify hit and miss times for cache management instructions that require conditional bus activity.
Cycle times marked with a : specify cycles of total latency and throughput.
Load and store multiple and string instruction cycles are shown as a fixed number of cycles plus a variable number of cycles where *n* is the number of words accessed by the instruction.

Chapter 8
Core Interface Operation

This chapter provides a general description of the coherent system bus (CSB), which is the interface between the core and the integrating device. Because most of the behavior of the CSB is not directly programmable, or even visible, to the user, this chapter does not attempt to describe all aspects of the CSB or even the most important CSB signals.

Instead, it describes mostly those aspects of the CSB that are configurable or that provide status information through the programming interface. It provides a glossary of those signals that are referenced in other chapters to offer a clearer understanding of how the core is integrated as part of a larger device.

NOTE

A bar over a signal name indicates that the signal is active-low—for example, *hreset* (hardware reset) and *int* (external interrupt). Active-low signals are referred to as asserted (active) when they are low and negated when they are high. Signals that are not active-low, such as *tben* (time base enable) and *ckstp* (checkstop interrupt) are referred to as asserted when they are high and negated when they are low.

8.1 Signal Groupings

A subset of the selected internal e300 coherent system bus (CSB) core signals is grouped as follows:

- Interrupts/resets—These signals include the external interrupt signal, checkstop signals, performance monitor interrupt (on e300c3), and both soft reset and hard reset signals. They are used to interrupt and, under various conditions, to reset the core.
- JTAG/debug interface signals—The JTAG (IEEE 1149.1-compliant) interface and debug unit provides a serial interface to the system for performing monitoring and boundary tests.
- Core status and control—These signals include the memory reservation signal, machine quiesce control signals, time base enable signal, and the *tlbisync* signal.
- Clock control—These signals provide for system clock input and frequency control.
- Test interface signals—Signals such as address matching, combinational matching, and watchpoint are used in the core for production testing.
- Transfer attribute signals—These signals provide information about the type of transfer, such as the transfer size and whether the transaction is bursted, write-through, or cache-inhibited.

8.1.1 Functional Groupings

Figure 8-1 shows the e300 core signals groups in greater detail. Note that the pm_event_in signal is only found on the e300c3.

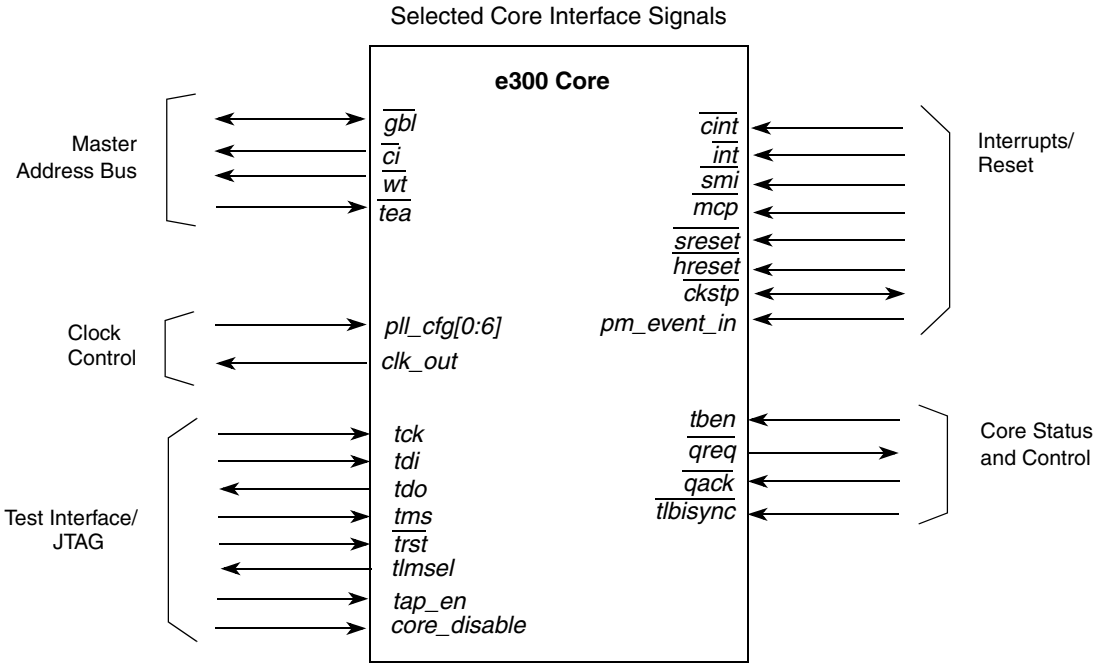


Figure 8-1. Core Interface Signals

8.1.2 Signal Summary

Table 8-1 provides alphabetically-ordered e300 core signals with related cross-references that are relevant to the user. It details the signal name, signal grouping, number of signals, and whether the signal is an input or an output.

Table 8-1. Summary of Selected Internal Signals

Signal	I/O	Comments, or Meaning when Asserted
Bus Signals: Master Address Bus		
$\overline{ci}$	O	Cache inhibit. Normally reflected from the I bit of the WIMG bits (regardless of whether the cache is enabled). For burst writes and address-only transactions, $\overline{ci}$ is always negated.
$\overline{gbl}$	I/O	Global. Normally reflected from the M bit of the WIMG bits; asserted indicates transaction is enabled for snooping by other masters. For burst writes, always negated. For address-only transactions that bypass translation, always asserted.
$\overline{wt}$	O	Write-through. Assertion indicates that a single-beat transaction is write-through, reflecting the value of the W bit of the WIMG bits for the block or page that contains the address of the current transaction.

Table 8-1. Summary of Selected Internal Signals (continued)

Signal	I/O	Comments, or Meaning when Asserted
tea	I	Transfer error acknowledge. Indicates that a data bus error occurred on the core interface. Causes a machine check interrupt (and possibly causes the processor to enter checkstop state if machine check enable bit is cleared (MSR[ME] = 0)).
External Interrupts		
hreset	I	Hard reset. Assertion resets the core.
sreset	I	Soft reset. When *sreset* is asserted, the processor attempts to reach a recoverable state by allowing the next instruction to either complete or cause an interrupt, blocking the completion of subsequent instructions and allowing the completed store queue to drain. Unlike a hard reset, no registers or latches are initialized.
int	I	External interrupt. Initiates an external interrupt to the core.
cint	I	Critical interrupt. Initiates a critical interrupt to the core.
mcp	I	Machine check interrupt. Indicates that the e300 core should initiate a machine check interrupt or enter the checkstop state as directed by the MSR.
smi	I	System management interrupt. If *smi* is asserted and MSR[EE] is set, the core initiates a system management interrupt.
ckstp	I/O	Checkstop interrupt. Assertion of this signal by the core is used to generate a chip-wide hard stop.
pm_event_in	I	Performance monitor interrupt. Initiates a performance monitor interrupt to the core.
Core Status		
tben	I	Asserted by the system logic to enable the time base.
qack	I	Quiescent acknowledge. Assertion Indicates that all bus activity that requires snooping has terminated or paused, and that the core may enter the quiescent (or low-power) state.
qreq	O	Quiescent request. Indicates that the core is requesting all CSB activity normally required to be snooped to terminate or to pause so the core may enter the quiescent (low-power) state. Once the core enters a quiescent state, it no longer snoops CSB activity.
tlbisync	I	TLB instruction synchronize. If asserted, the **tlbsync** instruction causes instruction execution to stop. If negated, instruction execution may continue or resume after the completion of a **tlbsync** instruction.
Clocks		
pll_cfg[0:6]	I	PLL configuration select. Configurations are as shown in Table 8-2.
clk_out	O	Assertion provides PLL clock output for PLL testing and monitoring. The *clk_out* signal clocks at either the core clock frequency or bus clock frequency, if enabled by the appropriate HID0 bits.
JTAG and TAP		
trst	I	JTAG test reset. This input causes asynchronous initialization of the internal JTAG test access port controller.
tck	I	JTAG test clock. Driven by a free-running clock signal. Input signals to the test access port are sampled on the rising edge of *tck*. TAP output signal changes occur on the falling edge of *tck*. The test logic allows TCK to be stopped asynchronously with respect to all other core complex clocks.

Table 8-1. Summary of Selected Internal Signals (continued)

Signal	I/O	Comments, or Meaning when Asserted
tms	I	JTAG test mode select. Decoded by the internal JTAG TAP controller to determine the primary operation of the test support circuitry.
tdi	I	JTAG test data input. The value present on the rising edge of *tck* is loaded into the selected JTAG test instruction or data register.
tdo	O	JTAG test data output. The contents of the selected internal instruction or data register are shifted out onto this signal on the falling edge of *tck*.
Test Interface		
tlmsel	O	TLM selected. *tlmsel* provides feedback to the external TAP linking module logic.
tap_en	I	TAP enable. *tap_en* is used by the TAP linking module (TLM) logic external to the core complex.
core_disable	I	On assertion, core output signals are negated or forced to a high-impedance state. The core enters a sleep mode, and instruction fetching and dispatching are disabled.

8.1.2.1 PLL Configuration (*pll_cfg*[0:6])—Input

The PLL is configured by *pll_cfg*[0:6]. For a given bus frequency, the PLL configuration signals set the internal CPU frequency of operation. Table 8-2 shows the PLL configuration options.

Table 8-2. Core PLL Configuration

PLL_CFG (0:1)	PLL_CFG (2:5)	PLL_CFG (6)	Bus-to-Core Multiplier	VCO divider
xx	0000	x	bypass/off	n/a
xx	1111	x	off	n/a
00	0001	0	1x	2
01	0001	0	1x	4
10	0001	0	1x	8
11	0001	0	1x	8
00	0001	1	1.5x	2
01	0001	1	1.5x	4
10	0001	1	1.5x	8
11	0001	1	1.5x	8
00	0010	0	2x	2
01	0010	0	2x	4
10	0010	0	2x	8
11	0010	0	2x	8
00	0010	1	2.5x	2
01	0010	1	2.5x	4

Table 8-2. Core PLL Configuration (continued)

PLL_CFG (0:1)	PLL_CFG (2:5)	PLL_CFG (6)	Bus-to-Core Multiplier	VCO divider
10	0010	1	2.5x	8
11	0010	1	2.5x	8
00	0011	0	3x	2
01	0011	0	3x	4
10	0011	0	3x	8
11	0011	0	3x	8
00	0011	1	3.5x	2
01	0011	1	3.5x	4
10	0011	1	3.5x	8
11	0011	1	3.5x	8
00	0100	0	4x	2
01	0100	0	4x	4
10	0100	0	4x	8
11	0100	0	4x	8
00	0100	1	4.5x	2
01	0100	1	4.5x	4
10	0100	1	4.5x	8
11	0100	1	4.5x	8
00	0101	0	5x	2
01	0101	0	5x	4
10	0101	0	5x	8
11	0101	0	5x	8
00	0101	1	5.5x	2
01	0101	1	5.5x	4
10	0101	1	5.5x	8
11	0101	1	5.5x	8
00	0110	0	6x	2
01	0110	0	6x	4
10	0110	0	6x	8
11	0110	0	6x	8
00	0110	1	6.5x	2
01	0110	1	6.5x	4

Table 8-2. Core PLL Configuration (continued)

PLL_CFG (0:1)	PLL_CFG (2:5)	PLL_CFG (6)	Bus-to-Core Multiplier	VCO divider
10	0110	1	6.5x	8
11	0110	1	6.5x	8
00	0111	0	7x	2
01	0111	0	7x	4
10	0111	0	7x	8
11	0111	0	7x	8
00	0111	1	7.5x	2
01	0111	1	7.5x	4
10	0111	1	7.5x	8
11	0111	1	7.5x	8
00	1000	0	8x	2
01	1000	0	8x	4
10	1000	0	8x	8
11	1000	0	8x	8
00	1000	1	8.5x	2
01	1000	1	8.5x	4
10	1000	1	8.5x	8
11	1000	1	8.5x	8
00	1001	0	9x	2
01	1001	0	9x	4
10	1001	0	9x	8
11	1001	0	9x	8
00	1001	1	9.5x	2
01	1001	1	9.5x	4
10	1001	1	9.5x	8
11	1001	1	9.5x	8
00	1010	0	10x	2
01	1010	0	10x	4
10	1010	0	10x	8
11	1010	0	10x	8
00	1010	1	10.5x	2
01	1010	1	10.5x	4

Table 8-2. Core PLL Configuration (continued)

PLL_CFG (0:1)	PLL_CFG (2:5)	PLL_CFG (6)	Bus-to-Core Multiplier	VCO divider
10	1010	1	10.5x	8
11	1010	1	10.5x	8
00	1011	0	11x	2
01	1011	0	11x	4
10	1011	0	11x	8
11	1011	0	11x	8
00	1011	1	11.5x	2
01	1011	1	11.5x	4
10	1011	1	11.5x	8
11	1011	1	11.5x	8
00	1100	0	12x	2
01	1100	0	12x	4
10	1100	0	12x	8
11	1100	0	12x	8
00	1101	0	16x	2
01	1101	0	16x	4
10	1101	0	16x	8
11	1101	0	16x	8
00	1110	0	20x	2
01	1110	0	20x	4
10	1110	0	20x	8
11	1110	0	20x	8
xx	all other modes	x	off	n/a

8.2 Overview of Core Interface Accesses

The e300 core contains an internal coherent system bus (CSB) that interfaces the core to the peripheral logic. This internal bus is very similar in function to the external 60x bus interface on the MPC603e processor. The CSB system logic decodes e300 core-initiated transactions and directs all accesses to the appropriate on-chip interface. The core interface prioritizes requests for bus operations from the instruction and data caches and performs bus operations following the coherent system bus (CSB) protocol.

The core interface includes address register queues, prioritization logic, and the bus control unit. The core interface latches snoop addresses for snooping in the data cache and address register queues, and reservations controlled by the Load Word and Reserve Indexed (**lwarx**) and Store Word Conditional

Indexed (**stwcx.**) instructions; it also maintains the touch load address for the data cache. The interface allows one level of pipelining; that is, with certain restrictions described in subsequent sections, there can be as many as two outstanding transactions at any given time. The e300 core also offers an optional pipeline extension to one-and-a-half level pipelining, which means that a new transaction can complete an address tenure when the previous transaction has been granted to the data bus; for the G2_LE core, a new transaction must wait until the previous data tenure has completed before completing its address tenure. Accesses are prioritized with load operations preceding store operations.

Instructions are automatically fetched from the memory system into the instruction unit where they are dispatched to the execution units or forwarded to the branch processing unit at a peak rate of three instructions per clock. Conversely, load and store instructions explicitly specify the movement of operands to and from the general-purpose and floating-point registers (GPRs and FPRs) and the memory system.Note that the e300c2 core does not support floating-point registers

When the e300 core encounters an instruction or data access, it calculates the logical address (effective address) and uses the low-order address bits to check for a hit in the on-chip instruction or data caches. During cache lookup, the instruction and data memory management units (MMUs) use the higher-order address bits to calculate the virtual address, allowing them to calculate the physical address (real address). The physical address bits are then compared with the corresponding cache tag bits to determine if a cache hit occurred. If the access misses in the corresponding cache, the physical address is used to access system memory.

In addition to loads, stores, and instruction fetches, the core performs software table search operations following TLB misses, cache cast-out operations when pseudo least recently used (PLRU) cache lines are written to memory after a cache miss, and cache-line snoop push-out operations when a modified cache line experiences a snoop hit from another bus master.

The core uses separate address and data buses and a variety of control and status signals for performing reads and writes. The address bus is 32 bits wide and the data bus is 64 bits wide. The bus can run at the full processor-clock frequency or at an integer division of the processor-clock speed. The implementation of the internal voltage of the e300 core is process-dependent; all I/O signals for the device depend on the device implementation. Note that the e300 core has no direct external I/O connection.

8.3 Interrupt, Checkstop, and Reset Signals

This section describes external interrupts, checkstop operations, and hard and soft reset inputs.

8.3.1 External Interrupts

Assertion of the external interrupt input signals (*$\overline{int}$*, *$\overline{smi}$*, *pm_event_in* (in e300c3) and *$\overline{mcp}$*) of the core eventually forces the processor to take an external interrupt, or a system management interrupt (*$\overline{smi}$)* if MSR[EE] is set, or performance monitor interrupt, or the machine check interrupt if MSR[ME] and HID0[EMCP] are set.

8.3.2 Checkstops

The e300 core has two checkstop input signals—*ckstp_in* (non-maskable) and *mcp* (enabled when MSR[ME] is cleared and HID0[EMCP] is set)—and a checkstop output (*ckstp_out*). If *ckstp_in* or *mcp* is asserted, the core halts operations by gating off all internal clocks. The core asserts *ckstp_out* if *ckstp_in* is asserted.

If *ckstp_out* is asserted by the core, it has entered the checkstop state and processing has halted internally. The *ckstp_out* signal can be asserted for various reasons including receiving a *tea* signal and detection of external parity errors. For more information about the checkstop state, see Section 5.5.2.2, "Checkstop State (MSR[ME] = 0)."

8.3.3 Reset Inputs

The e300 core has two reset inputs, described as follows:

- *hreset* (hard reset)—*hreset* is used for power-on reset sequences, or for situations in which the core must go through the entire cold-start sequence of internal hardware initializations.
- *sreset* (soft reset)—The soft reset input provides warm reset capability. This input can be used to avoid forcing the core to complete the cold start sequence.

When either reset input is negated, the processor attempts to fetch code from the system reset interrupt vector. The vector is located at offset 0x100 from the interrupt prefix (all zeros or ones, depending on the setting of the interrupt prefix bit, MSR[IP]. The IP bit is set on assertion of *hreset*.

8.3.4 Core Quiesce Control Signals

The core quiesce control signals (*qreq* and *qack* allow the processor to enter a low-power state and bring bus activity to a quiescent state in an orderly fashion.

The system quiesce state is entered by configuring the processor to assert the *qreq* output. This signal allows the system to terminate or pause any bus activities that are normally snooped. When the system is ready to enter the system quiesce state, it asserts *qack*. At this time, the core may enter a quiescent (low-power) state during which it stops snooping bus activity.

8.4 IEEE 1149.1-Compliant Interface

The e300 core boundary-scan interface is a fully-compliant implementation of the IEEE 1149.1 standard. This section describes the core IEEE 1149.1 (JTAG) interface.

8.4.1 IEEE 1149.1 Interface Description

The e300 core has five dedicated JTAG signals (described in Table 8-1). The *tdi* and *tdo* scan ports are used to scan instructions, as well as data, into the various scan registers for JTAG operations. The scan operation is controlled by the test access port controller, which is controlled by the *tms* input sequence. The scan data is latched in at the rising edge of *tck*.

Test reset (*trst*) is a JTAG optional signal used to reset the TAP controller asynchronously. The *trst* signal assures that the JTAG logic does not interfere with the normal operation of the device; this signal can be asserted concurrently with the assertion of *hreset*.

The e300 core implements the JTAG/debug in the same manner as does the G2 core with the exception of the 33-bit Run_N counter register in which the most-significant 32 bits form a 32-bit counter. The function of the least-significant bit remains unchanged. The Run_N counter is used by the debug functions to control the number of processor cycles that the processor runs before halting.

Chapter 9
Power Management

The e300 core is specifically designed for low-power operation. It provides both automatic and program-controllable power reduction modes for progressive reduction of power consumption. This chapter describes the hardware support provided by the e300 core for power management.

9.1 Overview

The e300 core has explicit power management features that are described in this chapter. Note that the design of the core is fully static, allowing the internal processor core state to be preserved when no internal clock is present.

The device drivers must be modified for power management, because operating systems service I/O requests by system calls to the device drivers. When a device driver is called to reduce the power of a device, it needs to be able to check the power state of the device, save the device configuration parameters, and put the device into a power-saving mode. Furthermore, every time the device driver is called, it needs to check the power status of the device and restore the device to the full-on state, if the device is in a power-saving mode.

9.2 Dynamic Power Management

Dynamic power management (DPM) automatically powers up and down the individual execution units of the core, based on the contents of the instruction stream. For example, if no floating-point instructions are being executed, the floating-point unit is automatically powered down.Note that floating-point instructions are not supported in the e300c2. Power is not actually removed from the execution unit; instead, each execution unit has an independent clock input, which is automatically controlled on a clock-by-clock basis. Because CMOS circuits consume negligible power when they are not switching, stopping the clock to an execution unit effectively eliminates its power consumption. The operation of DPM is completely transparent to software or any external hardware. Dynamic power management is enabled by setting HID0[DPM] on power-up following a hard reset sequence (*hreset*).

9.3 Programmable Power Modes

Hardware can enable a power management state through external asynchronous interrupts. The hardware interrupt causes the transfer of program flow to interrupt handler code. The appropriate mode is then set by the software. The core provides a separate interrupt and interrupt vector for power management: the system management interrupt (*smi*). The e300 core also contains a decrementer timer that allows it to enter the nap or doze mode for a predetermined period and then return to full power operation through the decrementer interrupt.

The core provides four power modes selectable by setting the appropriate control bits in the MSR and HID0. The four power modes are described briefly as follows:

- Full-power—This is the default power state of the core. The core is fully powered and the internal functional units are operating at the full processor clock speed. If the dynamic power management mode is enabled, functional units that are idle will automatically enter a low-power state without affecting performance, software execution, or external hardware.
- Doze—All the functional units of the core are disabled except for the time base/decrementer registers and the bus snooping logic. When the processor is in doze mode; an external asynchronous interrupt, system management interrupt, decrementer interrupt, hard or soft reset, or machine check input (*mcp*) brings the core into the full-power state. The core in doze mode maintains the phase-locked loop (PLL) in a full-power state and locked to the system external clock input (*sysclk*), so a transition to the full-power state takes only a few processor clock cycles.
- Nap—The nap mode further reduces power consumption by disabling bus snooping, leaving only the time base register and the PLL in a powered state. The core returns to the full-power state upon receipt of an external asynchronous interrupt, system management interrupt, decrementer interrupt, hard or soft reset, or machine check input (*mcp*) signal. A return to full-power state from a nap state takes only a few processor clock cycles.
- Sleep—Sleep mode reduces power consumption to a minimum by disabling all internal functional units; then external system logic may disable the PLL and *sysclk*. Returning the core to the full-power state requires the enabling of the PLL and *sysclk*, followed by the assertion of an external asynchronous interrupt, system management interrupt, hard or soft reset, or *mcp* signal after the time required to relock the PLL.

Note that the core cannot switch from one power management mode to another without first returning to full-on mode. The nap and sleep modes disable bus snooping; therefore, a hardware handshake using *qreq* and *qack* is provided to ensure coherency before the core enters these power management modes. Table 9-1 summarizes the four power states for the core.

Table 9-1. e300 **Core Programmable Power Modes**

PM Mode	Functioning Units	Activation Method	Full-Power Wake-Up Method
Full power	All units active	—	—
Full power (with DPM)	Requested logic by demand	By instruction dispatch	—
Doze	• Bus snooping • Data cache as needed • Decrementer timer	Controlled by SW	External asynchronous interrupts Decrementer interrupt Reset
Nap	Decrementer timer	Controlled by hardware and software	External asynchronous interrupts Decrementer interrupt Reset
Sleep	None	Controlled by hardware and software	External asynchronous interrupts Reset

9.3.1 Power Management Modes

The following sections describe the characteristics of the e300 core power management modes, the requirements for entering and exiting the various modes, and the system capabilities provided by the core while the power management modes are active.

9.3.1.1 Full-Power Mode with DPM Disabled

Full-power mode with DPM disabled is selected when the DPM enable bit in HID0[DPM] is cleared. The following characteristics apply:

- Default state following power-up and $\overline{hreset}$
- All functional units are operating at full processor speed at all times.

9.3.1.2 Full-Power Mode with DPM Enabled

Full-power mode with DPM enabled (HID0[DPM] = 1) provides on-chip power management without affecting the functionality or performance of the core as follows:

- Required functional units are operating at full processor speed
- Functional units are clocked only when needed
- No software or hardware intervention required after mode is set
- Software/hardware and performance transparent

9.3.1.3 Doze Mode

Doze mode disables most functional units but maintains cache coherency by enabling the bus interface unit and snooping. A snoop hit causes the core to enable the data cache, copy the data back to memory, disable the cache, and fully return to the doze mode.

Doze mode is characterized by the following features:

- Most functional units disabled
- Bus snooping and time base/decrementer still enabled
- PLL running and locked to internal *sysclk*

To enter the doze mode, the following conditions must occur:

- Set doze bit (HID0[8] = 1), MSR[POW] is set
- e300core enters doze mode after several processor clocks.

To return to full-power mode, the following conditions must occur:

- Assert internal $\overline{int}$, $\overline{smi}$, or $\overline{mcp}$ signals or decrementer interrupts
- Hard reset or soft reset
- Transition to full-power state occurs only after a few processor cycles.

9.3.1.4 Nap Mode

The nap mode disables the core except for the processor PLL and time base/decrementer. The time base can be used to restore the core to a full-on state after a specified period.

Because bus snooping is disabled for nap and sleep mode, a hardware handshake using the quiesce request ($\overline{qreq}$) and quiesce acknowledge ($\overline{qack}$) signals are required to maintain data coherency. The core asserts the $\overline{qreq}$ signal to indicate that it is ready to disable bus snooping, including all bus activity. Once the processor has entered a quiescent state, it no longer snoops bus activity.

When the system logic has ensured that snooping is no longer necessary, it allows the processor to enter the nap (or sleep) mode and causes the assertion of the core $\overline{qack}$ input signal for the duration of the nap mode period.

Nap mode is characterized by the following features:

- Time base/decrementer still enabled
- Most functional units disabled (including bus snooping)
- PLL running and locked to internal *sysclk*

To enter the nap mode, the following conditions must occur:

- Set nap bit (HID0[9] = 1), MSR[POW] bit is set
- e300 core asserts $\overline{qreq}$
- System asserts $\overline{qack}$
- The processor core enters nap mode after several processor clocks

To return to full-power mode, one of the following conditions must occur:

- Assert $\overline{int}$, $\overline{smi}$, or $\overline{mcp}$ internal signals
- Decrementer interrupt
- Hard reset or soft reset

Transition to full-power takes only a few processor cycles. $\overline{qack}$ can remain asserted; however, $\overline{qreq}$ negates before any bus transaction begins.

9.3.1.5 Sleep Mode

Sleep mode consumes the least amount of power of the four modes, since all functional units are disabled. To conserve the maximum amount of power, the PLL and internal *sysclk* signals can be disabled. Due to the fully static design of the e300 core, the internal processor state is preserved when no internal clock is present. Because the time base and decrementer are disabled while the core is in sleep mode, the time base contents must be updated from an external time base following sleep mode if accurate time-of-day maintenance is required.

Before entering sleep mode, the core asserts $\overline{qreq}$ to indicate that it is ready to disable bus snooping. When the system has ensured that snooping is no longer necessary, the system logic allows the core to enter sleep mode by asserting $\overline{qack}$ for the duration of the sleep mode period.

Sleep mode is characterized by the following features:

- All functional units disabled (including bus snooping and time base)
- All nonessential input receivers disabled
- Internal clock regenerators disabled
- PLL and *sysclk* can be disabled

To enter sleep mode, the following conditions must occur:

- Set sleep bit (HID0[10] = 1), MSR[POW] is set
- e300 core asserts $\overline{qreq}$
- System logic asserts $\overline{qack}$
- e300 core enters sleep mode after several processor clocks

To return to full-power mode when *sysclk* and PLL are not disabled, the following conditions must occur:

- Assert $\overline{int}$, $\overline{smi}$, or $\overline{mcp}$ internal signals
- Hard reset or soft reset

To return to full-power mode after PLL and *sysclk* are disabled in sleep mode, the following conditions must occur:

- Enable *sysclk*
- Reconfigure PLL into desired processor clock mode
- System logic waits for PLL startup and relock time (100 µsec)
- System logic asserts one of the sleep recovery signals (for example, $\overline{int}$ or $\overline{smi}$)

9.3.2 Power Management Software Considerations

Because the e300 core is a dual-issue processor core with out-of-order execution capability, care must be taken in how the power management modes are entered. Furthermore, nap and sleep modes require all outstanding bus operations to be completed before the power management mode is entered. Section 9.4, “Example Code Sequence for Entering Processor Sleep Mode,” provides an example software sequence for putting the e300 core into sleep mode.

Normally, during system configuration time, one of the power management modes is selected by setting the appropriate HID0 mode bit. Later, the power management mode is invoked by setting MSR[POW]. To ensure a clean transition into and out of the power management mode, set MSR[EE] (external interrupt enable) and execute the following code sequence:

```
        sync
        mtmsr[POW = 1]
        isync
loop:   b loop
```

9.4 Example Code Sequence for Entering Processor Sleep Mode

The following is a sample code sequence for entering e300 core sleep mode.

```
*****************************************************************
# set up e300 core HID0 power management bits
#****************************************************************
#******processor HID and external interrupt initialization*******************
#
# set up HID registers for the various processors of this family
# hid setup taken from minix's mpxPowerPC.s

        mfspr   r31, pvr         # pvr reg
        srawi    r31, r31, 16
resetTest603:
        cmpi    0, 0, r31, 3
        bne     cr0, endHIDSetup

        addi    r0, r0, 0
        oris    r0, r0, 0x8000   # enable machine check pin EMCP
        oris    r0, r0, 0x0010   # enable dynamic power mgmt DPM
        oris    r0, r0, 0x0020   # enable SLEEP power mode
        ori     r0, r0, 0x8000   # enable the Icache ICE
        ori     r0, r0, 0x4000   # enable the Dcache DCE
        ori     r0, r0, 0x0800   # invalidate Icache ICFI
        ori     r0, r0, 0x0400   # invalidate Dcache DCFI
        mtspr   hid0, r0
        isync
#****************************************************************
# then when the processor is in a loop, force an SMI interrupt
#****************************************************************

.orig 0x00001400                            # System Management Interrupt

# force big-endian mode
        stw         r0,0x05f8,r0            # need nop every second inst.
        stw         r0,0x05fc,r0
        mfmsr       r0
        ori         r0,r0,r0
        ori         r0,r0,0x0001            # force big-endian LE bit
        ori         r0,r0,r0
        xori        r0,r0,0x0001            # force big-endian LE bit
        ori         r0,r0,r0
        mtmsr       r0
        ori         r0,r0,r0
        isync
        ori         r0,r0,r0

# save off additional registers to be corrupted
        stw         r20,0x05f4,r0
        mfspr       r21, srr0               # put srr0 in r21
        stw         r21,0x05f0,r0           # put r21 in 0x05f0
        mfspr       r22, srr1               # put srr1 in r22
        stw         r22,0x05ec,r0           # put r22 in 0x05ec
        stw         r23,0x05e8,r0
        mfcr        r23
        stw         r23,0x05e4,r0
        xor         r0,r0,r0
```

```
#*****************************************************************
# set msr pow bit to go into sleep mode

        sync
        mfmsr r5                            # get MSR
        addis r3, r0, 0x0004                # turn on POW bit
        ori   r3, r3, 0x0000                # turn on ME bit 19
        or    r5, r3, r5
        mtmsr r5
        isync

        addis r20, r0, 0x0000
        ori   r20, r20, 0x0002
stay_here:
        addic.  r20, r20, -1                # subtract 1 from r20 and set cc
        bgt cr0, stay_here                  # loop if positive


# restore corrupted registers
        lwz         r23,0x05e4,r0
        mtcrf       0xff,r23
        lwz         r23,0x05e8,r0
        lwz         r22,0x05ec,r0
        mtspr       srr1, r22
        lwz         r21,0x05f0,r0
        mtspr       srr0, r21
        lwz         r20,0x05f4,r0
        lwz         r0,0x05fc,r0

        sync
        rfi

#*****************************************************************
# to get out of sleep mode, do a Soft Reset
#*****************************************************************

.orig 0x00000100                            # Reset handler in low memory

# force big-endian mode
        stw         r0,0x05f8,r0            # need nop every second inst.
        stw         r0,0x05fc,r0
        mfmsr       r0
        ori         r0,r0,r0
        ori         r0,r0,0x0001            # force big-endian LE bit
        ori         r0,r0,r0
        xori        r0,r0,0x0001            # force big-endian LE bit
        ori         r0,r0,r0
        mtmsr       r0
        ori         r0,r0,r0
        isync
        ori         r0,r0,r0

# save off additional registers to be corrupted
        stw         r20,0x05f4,r0
        stw         r21,0x05f0,r0
        stw         r22,0x05ec,r0
```

```
        stw         r23,0x05e8,r0
        mfcr        r23
        stw         r23,0x05e4,r0
        xor         r0,r0,r0

# restore corrupted registers
        lwz         r23,0x05e4,r0
        mtcrf       0xff,r23
        lwz         r23,0x05e8,r0
        lwz         r22,0x05ec,r0
        lwz         r21,0x05f0,r0
        lwz         r20,0x05f4,r0
        lwz         r0,0x05fc,r0

        sync
        rfi
#*****************************************************************
```

Chapter 10
Debug Features

This chapter describes the debug features of the PowerPC architecture with respect to the e300 core. The e300 includes the trace facility debug features. The enhanced debug features are described as follows:

- Addition of breakpoint status bits to IBCR and DBCR

10.1 Breakpoint Resources

The e300 core provides enhanced debug facilities—instruction address breakpoint, data address breakpoint, and program single stepping to enable software debug events. The original IABR and single-step functions are supplemented by the new debug features. The debug facilities consist of a set of debug control registers (DBCR, IBCR), a set of instruction address breakpoint registers (IABR, IABR2), and a set of data address breakpoint registers (DABR, DABR2). These registers used together enable various breakpoint functions. Additional hardware debug facilities exist through the JTAG/debug interface.

These registers are accessible only to supervisor-level programs by the **mfspr** and **mtspr** instructions. The SPR addresses for the registers can be found in Table 3-32 of Chapter 3, "Instruction Set Model."

When an instruction or data address breakpoint register is enabled and the conditions are met, an instruction address breakpoint interrupt (0x01300) or DSI interrupt (0x00300) occurs. The conditions for these exceptions are described in Section 10.1.6, "Interrupt Vectors for Debugging."

10.1.1 Instruction Address Breakpoint Registers (IABR, IABR2)

IABR and IABR2 can be used to cause a breakpoint interrupt if a specified instruction address is encountered. IABR and IABR2 control the instruction address breakpoint interrupt. IABR[CEA] and IABR2[CEA] hold the effective address to which each instruction's address is compared. The interrupt for each breakpoint is enabled by setting IABR[BE] or IABR2[BE], respectively. The interrupt is taken when there is an instruction address breakpoint match on the next instruction to complete. The instruction tagged with the match cannot complete before the instruction address breakpoint interrupt (0x01300) is taken. The address of the instruction that matches the breakpoint condition is stored in SRR0. The tagged instruction retires after returning from the interrupt (**rfi** or **rfci**). The results are then committed to the destination registers and address.

If the IABR or IABR2 values are set to any interrupt vector range, an unrecoverable state occurs. The IABR or IABR2 values should never be set to match within the instruction address breakpoint interrupt handler. Allowing a breakpoint within any handler may result in an indeterminate or unrecoverable processor state. See Section 2.2.14, "Instruction Address Breakpoint Registers (IABR and IABR2)," for bit descriptions.

10.1.2 Instructional Address Control Register (IBCR)

IBCR is a supervisor-level SPR. It controls the compare and match type conditions for IABR and IABR2. Note that IABR and IABR2 must be enabled before the effects of IBCR are realized. The e300 includes additional bits in IBCR[6–7] that contain the status of whether an instruction address breakpoint has matched. See Section 2.2.15, "Instruction Address Breakpoint Control Register (IBCR)," for bit descriptions.

10.1.3 Data Address Breakpoint Registers (DABR, DABR2)

The DABR and DABR2 registers are used to cause a breakpoint interrupt if the specified address is encountered for a data access. DABR[CEA] and DABR2[CEA] hold an effective address to which each address of a data access is compared. The breakpoint translation bit of DABR is also compared with MSR[DR] to check for a complete match. A match occurs when MSR[DR] = DABR[BT]. The data address write and data address read interrupts are enabled by setting DABR[WBE,RBE] and DABR2[WBE,RBE]. The data access tagged with the match does not complete before the breakpoint interrupt is taken.

The DSI interrupt (0x00300) occurs when there is a data address breakpoint match. The DSI interrupt is taken before the load or store instruction is executed. When the interrupt is taken, DAR is set to the data address that causes the breakpoint and DSISR[9] is set to indicate a data address breakpoint. The address of the instruction associated with the breakpoint condition is stored in SRR0. The instruction retires after returning from the DSI interrupt, and all registers and memory accesses are committed to memory.

An unrecoverable state occurs whenever DABR or DABR2 values are set to an interrupt vector. These values must not be set to match within the DSI interrupt handler or the core may enter an indeterminate or unrecoverable processor core state.

10.1.4 Data Address Control Register (DBCR)

DBCR is a supervisor-level SPR on the e300 core that controls the compare type and match type conditions for DABR and DABR2. Note that DABR or DABR2 (or both) must be enabled before the effects of DBCR are realized. The e300 core includes additional bits in DBCR[6–7] that contain the status of whether a data address breakpoint has matched. See Section 2.2.17, "Data Address Breakpoint Control Register (DBCR)," for bit descriptions.

10.1.5 Other Debug Resources

In addition to the four breakpoint registers and the two breakpoint control registers, other internal register values control and monitor the effects of breakpoint conditions. Table 10-1 shows these registers and their bits.

Table 10-1. Other Debug and Support Register Bits

Register	Bits	Name	Description
MSR	17	PR	Privilege level. Breakpoint registers can only be accessed when this bit is cleared (PR = 0 corresponds to supervisor mode).
	21	SE	Single-step trace enable. 0 The processor executes instructions normally. 1 The processor generates a trace interrupt upon the successful completion of the next instruction.
	22	BE	Branch trace enable 0 The processor executes branch instructions normally. 1 The processor generates a trace interrupt upon the successful completion of a branch instruction.
HID0	0–31	—	See Table 2-8 for details.
DAR	0–31	—	Data address register. DAR is loaded with the effective address of a data breakpoint condition that matches.
DSISR	9	DABR	Set if a DABR interrupt occurs.

10.1.6 Interrupt Vectors for Debugging

Table 10-2 lists the interrupt vectors that are associated with debug and breakpoint events. Breakpoint events do not change other interrupt vectors and conditions.

Table 10-2. Debug Interrupts and Conditions

Interrupt Type	Vector Offset	Exception Condition
Data access (DSI)	00300	A data access breakpoint interrupt occurs when a match condition exists for the effective address of the data access in either DABR or DABR2 for the next read or write data access, and the corresponding WBE or RBE, DABR enable bits are set for a read or write access, respectively. When a data breakpoint event occurs, DSISR[9] is set, identifying the DSI as having been caused by a data breakpoint event. In this case, the DAR contains the address of the data access that matched.
Trace	00D00	A trace interrupt is taken when MSR[SE] = 1 or when the currently completing instruction is a branch instruction and MSR[BE] = 1.
Instruction address breakpoint	01300	An instruction address breakpoint interrupt occurs when a match condition exists for the effective address of the instruction access in either IABR or IABR2 for the next instruction to complete in the completion unit, and the corresponding IABR[BE] enable bit is set.

10.2 Using Breakpoint Facilities

Breakpoints, single-stepping, branch tracing, address and combinational matching are debugging facilities provided by the breakpoint registers (DABR, DABR2, IABR, and IABR2) and the MSR.

10.2.1 Single-Stepping

Single-stepping can be a very useful tool in software debugging. This debug feature executes one instruction before it takes a trace interrupt. In the trace interrupt handler, the results of executing that instruction can be examined.

When MSR[SE] (single-step trace enable) is set, the processor generates a trace interrupt (0x00D00) upon the successful completion of the next instruction. A trace interrupt is not taken for an **isync**, **sync**, **rfi**, **rfci**, or **mtmsr** instructions.

MSR[SE] can be set by using **mtmsr** or by setting the SRR0 bit corresponding to MSR[SE] before returning from an interrupt. If MSR[SE] is set by restoring SRR0 to the MSR on the return from an interrupt, single-stepping is enabled and one instruction is executed, followed by a trace interrupt.

A typical software debugging procedure is to set an instruction address breakpoint at the instruction address to be single stepped. When the IABR interrupt is taken, the interrupt routine disables the instruction address breakpoint and sets SRR0 to set the MSR[SE] on the **rfi**. The trace interrupt is then taken upon the completion of the first instruction after return from the IABR interrupt. For any interrupt, the value of MSR is saved in SRR0. The value of MSR[SE] is automatically cleared within the interrupt handler, disabling single-stepping while the trace interrupt handler is executed. In this typical case, the trace interrupt handler can then examine the results of the execution of the instruction in question. The trace interrupt handler can then clear the appropriate bit in SRR0 to disable single-stepping (the bit in SRR0 that will cause MSR[SE] = 0) on the **rfi** if no more single-stepping is needed.

Single-stepping skips **isync**, **sync**, **rfi**, **rfci**, and branch instructions because these instructions do not enter the instruction pipeline. The branch trace feature, described in Section 10.2.2, “Branch Tracing,” may be used to single-step through **rfi**, **rfci**, and branch instructions.

Note that single-stepping on an **mtmsr** may give unwanted results. The new value moved into MSR upon execution of the **mtmsr** might cause single-stepping to be disabled (if the new value of MSR[SE] is cleared). Thus, it is recommended that the value of MSR[SE] be changed (to enable or disable single-stepping) indirectly, by changing the value of SRR0 within an interrupt handler and relying on **rfi** to set or clear MSR[SE].

10.2.2 Branch Tracing

When MSR[BE] (branch trace enable) is set, the processor generates a trace interrupt (0x00D00) upon the successful completion of a branch instruction.

10.2.3 Breakpoint Address Matching Options

When an instruction address breakpoint is set, and a condition is matched, an instruction address breakpoint interrupt (0x01300) occurs along with execution of the matched instruction. The instruction retires after the return from the interrupt handler. When a data address and data translation condition match occurs, a DSI interrupt (0x00300) occurs along with execution of the matched instruction. The instruction retires after the return from the interrupt handler occurs and the instruction has updated memory or registers, as appropriate.

On the e300, a match occurs when an address equals the effective address in a corresponding breakpoint register. The e300 can also match addresses on a greater than or equal to, or less than basis, as an additional matching condition for IBCR and DBCR.

An address match can be signaled after an OR function of the two compared addresses match or the AND of the two addresses match, depending on the setting of IBCR and DBCR. This feature along with

matching on greater than and less than allows a breakpoint to be set inside or outside a range of two addresses. The instruction address breakpoints and data address breakpoints always operate independently of each other. For more details, see Section 2.2.15, "Instruction Address Breakpoint Control Register (IBCR)," and Section 2.2.17, "Data Address Breakpoint Control Register (DBCR)."

The address matching for the instruction address breakpoint register has the following four possible conditions for the specific register:

1. Instruction's effective address = IABR[CEA].
   Table 10-3 describes the instruction address breakpoint register for a single address matching condition. In this case, only one IABR is used.

Table 10-3. Single-Address Matching Bit Settings

Register Field Name	Condition	Register Field Name	Condition
IABR[CEA]	—	IABR2[CEA]	—
IABR[BE]	1	IABR2[BE]	0
IBCR[CNT]	0	—	—
IBCR[SIG_TYPE]	OR	—	—
IBCR[CMP1]	=	IBCR[CMP2]	—

   With single address matching settings, a match occurs when the instruction's effective address = IABR[CEA].

2. Instruction's effective address = IABR[CEA] OR IABR2[CEA].
   Table 10-4 describes the instruction address breakpoint register settings when an address can match one or the other possible addresses (an OR condition). This requires both IABR registers to be programmed.

Table 10-4. Two-Address OR Matching

Register Field Name	Condition	Register Field Name	Condition
IABR[CEA]	—	IABR2[CEA]	—
IABR[BE]	1	IABR2[BE]	1
IBCR[CNT]	0	—	—
IBCR[SIG_TYPE]	OR	—	—
IBCR[CMP1]	=	IBCR[CMP2]	=

   With two address OR matching, a match occurs when the instruction's effective address = IABR[CEA] OR the instruction's effective address = IABR2[CEA].

3. IABR[CEA] < instruction's effective address < IABR2[CEA].
   Table 10-5 describes the instruction address breakpoint register settings for an address matching inside an address range condition.

Table 10-5. Address Matching for Inside Address Range

Register Field Name	Condition	Register Field Name	Condition
IABR[CEA]	—	IABR2[CEA]	—
IABR[BE]	1	IABR2[BE]	1
IBCR[CNT]	0	—	—
IBCR[SIG_TYPE]	AND	—	—
IBCR[CMP1]	>	IBCR[CMP2]	<

With address matching for an inside address range, a match occurs when IABR[CEA] ≤ instruction's effective address < IABR2[CEA].

4. Instruction's effective address < IABR[CEA] OR instruction's effective address > IABR2[CEA]. Table 10-6 describes the instruction address breakpoint register settings for an address matching outside an address range condition.

Table 10-6. Address Matching for Outside Address Range

Signal	Condition	Signal	Condition
IABR[CEA]	—	IABR2[CEA]	—
IABR[BE]	1	IABR2[BE]	1
IBCR[CNT]	0	—	—
IBCR[SIG_TYPE]	OR	—	—
IBCR[CMP1]	<	IBCR[CMP2]	>

With address matching for an outside address range, a match occurs when the instruction's effective address < IABR[CEA] OR the instruction's effective address ≤ IABR2[CEA].

For more details, see Section 2.2.15, "Instruction Address Breakpoint Control Register (IBCR)," and Section 2.2.17, "Data Address Breakpoint Control Register (DBCR)."

10.3 Synchronization Requirements and Other Precautions

An **isync** instruction must follow the execution of the **mtspr** to the breakpoint related registers, HID0, IABR, IABR2, DABR, DABR2, IBCR, and DBCR (or **mtmsr** for MSR) to ensure that the breakpoint condition is set. IBCR and DBCR should be set before a corresponding breakpoint is enabled. The breakpoint enable bits should be cleared before changing bits in the IBCR and DCBR. For more details, see Section 5.5.17, "Instruction Address Breakpoint Interrupt (0x01300)."

An unrecoverable state occurs at any time if one of the register values of IABR, IABR2, DABR, and DABR2 are set to point to an interrupt vector. The IABR or IABR2 values must not be set to match within the instruction address breakpoint interrupt handler, and the DABR or DABR2 values must not be set to match within the DSI interrupt handler. Setting a breakpoint within the instruction address breakpoint interrupt or DSI handler may result in an unrecoverable and indeterminate processor core state.

If an IABR match and DABR match occur on the same instruction, the instruction address breakpoint interrupt is taken before the DSI interrupt.

If an IABR match occurs on a branch instruction, the instruction address breakpoint interrupt is set to the effective address of the branch instruction.

Chapter 11
Performance Monitor

This chapter describes the performance monitor as implemented in the e300c3 core. The programming model is similar to that defined by the EIS, some features are defined by the e300c3 implementation, in particular, the events that can be counted.

11.1 Overview

The performance monitor provides the ability to count predefined events and processor clocks associated with particular operations, for example cache misses, mispredicted branches, or the number of cycles an execution unit stalls. The count of such events can be used to trigger the performance monitor interrupt.

The performance monitor can be used to do the following:

- Improve system performance by monitoring software execution and then recoding algorithms for more efficiency. For example, memory hierarchy behavior can be monitored and analyzed to optimize task scheduling or data distribution algorithms.
- Characterize processors in environments not easily characterized by benchmarking.
- Help system developers bring up and debug their systems.

The performance monitor uses the following resources:

- The performance monitor mark bit in the MSR (MSR[PMM]). This bit controls which programs are monitored.
- The move to/from performance monitor registers (PMR) instructions, **mtpmr** and **mfpmr**.
- The external input *pm_event_in*.
- PMRs:
  - The performance monitor counter registers (PMC0–PMC3) are 32-bit counters used to count software-selectable events. Each counter counts up to 128 events. UPMC0–UPMC3 provide user-level read access to these registers. Reference events are those that should be applicable to most microprocessor microarchitectures and be of general value. They are identified in Table 11-9.
  - The performance monitor global control register (PMGC0) controls the counting of performance monitor events. It takes priority over all other performance monitor control registers. UPMGC0 provides user-level read access to PMGC0.
  - The performance monitor local control registers (PMLCa0–PMLCa3) control each individual performance monitor counter. Each counter has a corresponding PMLCa register. UPMLCa0–UPMLCa3 provide user-level read access to PMLCa0–PMLCa3).
- The performance monitor interrupt is assigned to interrupt vector 0x0F00. Its priority is less than the fixed-interval interrupt and greater than the decrementer interrupt.

Software communication with the performance monitor is achieved through PMRs rather than SPRs. The PMRs are used for enabling conditions that can trigger the performance monitor interrupt.

11.2 Performance Monitor Registers

The performance monitor provides a set of PMRs for defining, enabling, and counting conditions that trigger the performance interrupt. It also the performance monitor interrupt vector.

The supervisor-level performance monitor registers in Table 11-1 are accessed with **mtpmr** and **mfpmr**. Attempting to read or write supervisor-level registers in user-mode causes a privilege exception.

Table 11-1. Performance Monitor Registers–Supervisor Level

Number	PMR[0–4]	PMR[5–9]	Name	Abbreviation
16	00000	10000	Performance monitor counter 0	PMC0
17	00000	10001	Performance monitor counter 1	PMC1
18	00000	10010	Performance monitor counter 2	PMC2
19	00000	10011	Performance monitor counter 3	PMC3
144	00100	10000	Performance monitor local control a0	PMLCa0
145	00100	10001	Performance monitor local control a1	PMLCa1
146	00100	10010	Performance monitor local control a2	PMLCa2
147	00100	10011	Performance monitor local control a3	PMLCa3
400	01100	10000	Performance monitor global control 0	PMGC0

The user-level performance monitor registers in Table 11-2 are read-only and are accessed with the **mfpmr** instruction. Attempting to write these user-level registers in either supervisor or user mode causes an illegal instruction exception.

Table 11-2. Performance Monitor Registers–User Level (Read-Only)

Number	PMR[0–4]	PMR[5–9]	Name	Abbreviation
0	00000	00000	Performance monitor counter 0	UPMC0
1	00000	00001	Performance monitor counter 1	UPMC1
2	00000	00010	Performance monitor counter 2	UPMC2
3	00000	00011	Performance monitor counter 3	UPMC3
128	00100	00000	Performance monitor local control a0	UPMLCa0
129	00100	00001	Performance monitor local control a1	UPMLCa1
130	00100	00010	Performance monitor local control a2	UPMLCa2
131	00100	00011	Performance monitor local control a3	UPMLCa3
384	01100	00000	Performance monitor global control 0	UPMGC0

11.2.1 Global Control Register 0 (PMGC0)

The performance monitor global control register (PMGC0), shown in Figure 11-1, controls all performance monitor counters.

PMGC0 (PMR400)
UPMGC0 (PMR384)

Access: PMGC0: Supervisor-only
UPMGC0: Supervisor/user read-only

	0	1	2	3	18	19	20	21	22	23	24	31
R W	FAC	PMIE	FCECE	—								
R W	FAC	PMIE	FCECE	—		TBSEL	—			TBEE	—	
Reset	All zeros											

Figure 11-1. Performance Monitor Global Control Register 0 (PMGC0)/ User Performance Monitor Global Control Register 0 (UPMGC0)

PMGC0 is cleared by a hard reset. Reading this register does not change its contents. Table 11-3 describes PMGC0 fields.

Table 11-3. PMGC0 Field Descriptions

Bits	Name	Description
0	FAC	Freeze all counters. When FAC is set by hardware or software, PMLCx[FC] maintains its current value until it is changed by software. 0 The PMCs are incremented (if permitted by other PM control bits). 1 The PMCs are not incremented.
1	PMIE	Performance monitor interrupt enable 0 Performance monitor interrupts are disabled. 1 Performance monitor interrupts are enabled and occur when an enabled condition or event occurs, at which time PMGC0[PMIE] is cleared Software can clear PMIE to prevent performance monitor interrupts. Performance monitor interrupts are caused by time base events or PMCx overflow.
2	FCECE	Freeze counters on enabled condition or event 0 The PMCs can be incremented (if permitted by other PM control bits). 1 The PMCs can be incremented (if permitted by other PM control bits) only until an enabled condition or event occurs. When an enabled condition or event occurs, PMGC0[FAC] is set. It is up to software to clear FAC. An enabled condition or event is defined as one of the following: • When the msb = 1 in PMC*x* and PMLCa*x*[CE] = 1. • When the time-base bit specified by TBSEL=1 and TBEE=1.
3–17	—	Reserved, should be cleared.

Table 11-3. PMGC0 Field Descriptions (continued)

Bits	Name	Description
19-20	TBSEL	Time base selector. Selects the time base bit that can cause a time base transition event (the event occurs when the selected bit changes from 0 to 1). 00 TB[63] (TBL[31]) 01 TB[55] (TBL[23]) 10 TB[51] (TBL[19]) 11 TB[47] (TBL[15]) Time base transition events can be used to periodically collect information about processor activity. In multiprocessor systems in which TB registers are synchronized across processors, these events can be used to correlate performance monitor data obtained by the several processors. For this use, software must specify the same TBSEL value for all processors in the system. Time-base frequency is implementation-dependent, so software should invoke a system service program to obtain the frequency before choosing a TBSEL value.
20–21	—	Reserved, should be cleared.
23	TBEE	Time base transition event exception enable 0 Exceptions from time base transition events are disabled. 1 Exceptions from time base transition events are enabled. A time base transition is signalled to the performance monitor if the TB bit specified in PMGC0[TBSEL] changes from 0 to 1. Time base transition events can be used to freeze counters (PMGC0[FCECE]) or signal an exception (PMGC0[PMIE]). Changing PMGC0[TBSEL] while PMGC0[TBEE] is enabled may cause a false 0 to 1 transition that signals the specified action (freeze, exception) to occur immediately. Although the interrupt signal condition may occur with MSR[EE] = 0, the interrupt cannot be taken until MSR[EE] = 1.
12-31	—	Reserved, should be cleared.

11.2.2 User Global Control Register 0 (UPMGC0)

The contents of PMGC0 are reflected to UPMGC0, which can be read by user-level software. UPMGC0 can be read with the **mfpmr** instruction using PMR384.

11.2.3 Local Control A Registers (PMLCa0–PMLCa3)

The local control A registers (PMLCa0–PMLCa3) function as event selectors and give local control for the corresponding performance monitor counters. PMLCa works with the corresponding PMLCb register. PMLCa registers are shown in Figure 11-2.

PMLCa0 (PMR144)
PMLCa1 (PMR145)
PMLCa2 (PMR146)
PMLCa3 (PMR147)

UPMLCa0 (PMR128)
UPMLCa1 (PMR129)
UPMLCa2 (PMR130)
UPMLCa3 (PMR131)

Access: PMLCa0–PMLCa3: Supervisor-only
UPMLCa0–UPMLCa3: Supervisor/user read-only

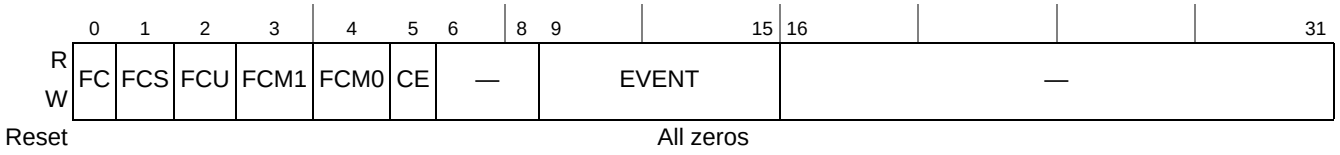


Figure 11-2. Local Control A Registers (PMLCa0–PMLCa3)/ User Local Control A Registers (UPMLCa0–UPMLCa3)

PMLCa registers are cleared by a hard reset. Table 11-4 describes PMLCa fields.

Table 11-4. PMLCa0–PMLCa3 Field Descriptions

Bits	Name	Description
0	FC	Freeze counter. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC cannot be incremented.
1	FCS	Freeze counter in supervisor state. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC cannot be incremented if MSR[PR] is cleared.
2	FCU	Freeze counter in user state. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC cannot be incremented if MSR[PR] is set.
3	FCM1	Freeze counter while mark is set. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC cannot be incremented if MSR[PMM] is set.
4	FCM0	Freeze counter while mark is cleared. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC cannot be incremented if MSR[PMM] is cleared.
537	CE	Condition enable. 0 Overflow conditions for PMC*n* cannot occur (PMC*n* cannot cause interrupts or freeze counters) 1 Overflow conditions occur when the most-significant-bit of PMC*n* is equal to 1. It is recommended that CE be cleared when counter PMC*n* is selected for chaining.
6-8	—	Reserved, should be cleared.
9–15	EVENT	Event selector. Up to 128 events selectable. See Section 11.5.2, “Event Selection”
16–31	—	Reserved, should be cleared.

11.2.4 User Local Control A Registers (UPMLCa0–UPMLCa3)

The PMLCa contents are reflected to UPMLCa0–UPMLCa3, which can be read by user-level software with **mfpmr** using PMR numbers in Table 11-2.

11.2.5 Performance Monitor Counter Registers (PMC0–PMC3)

The performance monitor counter registers (PMC0–PMC3), shown in Figure 11-3, are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count up to 128 events.

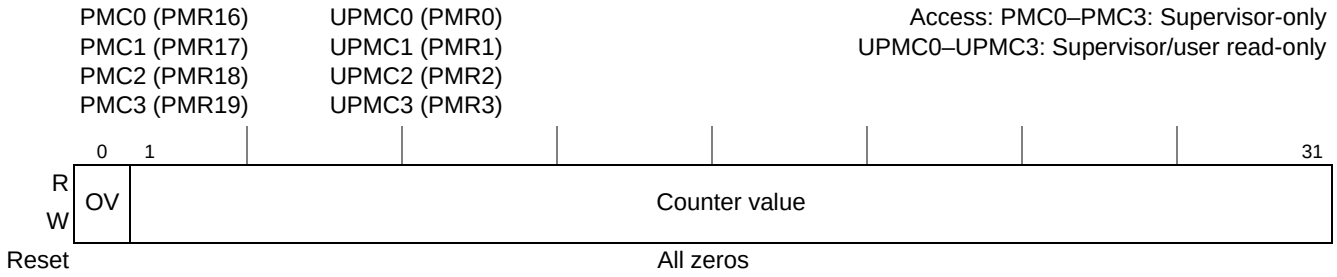


Figure 11-3. Performance Monitor Counter Registers (PMC0–PMC3)/ User Performance Monitor Counter Registers (UPMC0–UPMC3)

PMCs are cleared by a hard reset. Table 11-5 describes PMC register fields.

Table 11-5. PMC0–PMC3 Field Descriptions

Bits	Name	Description
0	OV	Overflow. 0 Counter has not reached an overflow state. 1 Counter has reached an overflow state.
1–31	Counter Value	Indicates the number of occurrences of the specified event.

The minimum counter value is 0x0000_0000; 4,294,967,295 (0xFFFF_FFFF) is the maximum. A counter can increment by 0, 1, or 2 up to the maximum value and then wraps to the minimum value.

A counter enters overflow state when the high-order bit is set by entering the overflow state at the halfway point between the minimum and maximum values. A performance monitor interrupt handler can easily identify overflowed counters, even if the interrupt is masked for many cycles (during which the counters may continue incrementing). A high-order bit is set normally only when the counter increments from a value below 2,147,483,648 (0x8000_0000) to a value greater than or equal to 2,147,483,648 (0x8000_0000).

NOTE

Initializing PMCs to overflowed values is strongly discouraged. If an overflowed value is loaded into a PMC*n* that held a non-overflowed value (and PMGC0[PMIE], PMLCa*n*[CE], and MSR[EE] are set), an interrupt is generated before any events are counted.

The response to an overflow depends on the configuration, as follows:

- If PMLCa*n*[CE] is clear, no special actions occur on overflow: the counter continues incrementing, and no exception is signaled.
- If PMLCa*n*[CE] and PMGC0[FCECE] are set, all counters are frozen when PMC*n* overflows.
- If PMLCa*n*[CE] and PMGC0[PMIE] are set, an exception is signaled when PMC*n* reaches overflow. Interrupts are masked by clearing MSR[EE]. An exception may be signaled while MSR[EE] is cleared, but the interrupt is not taken until it is set and only if the overflow condition is still present and the configuration has not been changed in the meantime to disable the exception.

However, if MSR[EE] remains clear until after the counter leaves the overflow state (msb becomes 0), or if MSR[EE] remains clear until after PMLCa*n*[CE] or PMGC0[PMIE] cleared, the exception is not signaled.

The following sequence is recommended for setting counter values and configurations:

1. Set PMGC0[FAC] to freeze the counters.
2. Using **mtpmr** instructions, initialize counters and configure control registers.
3. Release the counters by clearing PMGC0[FAC] with a final **mtpmr**.

11.2.6 User Performance Monitor Counter Registers (UPMC0–UPMC3)

The contents of PMC0–PMC3 are reflected to UPMC0–UPMC3, which can be read by user-level software with the **mfpmr** instruction using PMR numbers in Table 11-2.

11.2.7 Performance Monitor Instructions

The APU defines instructions for reading and writing the PMRs as shown in Table 11-6.

Table 11-6. Performance Monitor APU Instructions

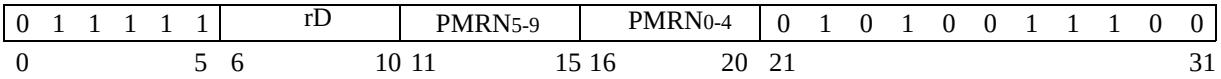
Name	Mnemonic	Syntax
Move from Performance Monitor Register	**mfpmr**	**r**D,PMRN
Move to Performance Monitor Register	**mtpmr**	PMRN,**r**S

The following instruction has been added to support performance monitor operations:

mfpmr
Move From Performance Monitor Register

mfpmr **rD, PMRN**

mfpmr
Integer Unit



GPR (rD) <-- PMREG(PMRN)

PMRN denotes a performance monitor register as listed in Table 11-1 and Table 11-2.

The contents of the designated performance monitor register are placed into GPR[rD].

When MSR[PR] = 1, specifying a performance monitor register that is not implemented and is not privileged (PMRN[5] = 0) results in an illegal instruction exception-type program interrupt. When MSR[PR]=1, specifying a performance monitor register that is privileged (PMRN[5] = 1) results in a privileged instruction execution-type program interrupt. When MSR[PR] = 0, specifying an unimplemented performance monitor register is boundedly undefined.

Other registers altered:

None

mtpmr
Move To Performance Monitor Register

mtpmr **PMRN, rS**

mtpmr
Integer Unit

0	1	1	1	1	1	rS	PMRN5-9	PMRN0-4	0	1	1	1	0	0	1	1	1	0	0
0					5	6	10	11	15	16	20	21							31

PMREG(PMRN) <-- GPR (rS)

PMRN denotes a performance monitor register, as listed in Table 11-1 and Table 11-2.

The contents of GPR[rS] are placed into the designated performance monitor register.

When MSR[PR] = 1, specifying a performance monitor register that is not implemented and is not privileged (PMRN[5] = 0) results in an illegal instruction exception-type program interrupt. When MSR[PR]=1, specifying a performance monitor register that is privileged (PMRN[5] = 1) results in a privileged instruction execution-type program interrupt. When MSR[PR] = 0, specifying an unimplemented performance monitor register is boundedly undefined.

Other registers altered:

none

11.3 Performance Monitor Interrupt

The performance monitor interrupt is triggered by an enabled condition or event. The only performance monitor enabled condition or event defined for the e300c3 is the following:

- A PMC*n* overflow condition occurs when both of the following are true:
  - The counter's overflow condition is enabled; PMLCa*n*[CE] is set.
  - The counter indicates an overflow; PMC*n*[OV] is set.

If PMGC0[PMIE] is set, an enabled condition or event triggers the signaling of a performance monitor exception.

If PMGC0[FCECE] is set, an enabled condition or event also triggers all performance monitor counters to freeze.

Although the performance monitor exception condition could occur with MSR[EE] cleared, the interrupt cannot be taken until MSR[EE] is set. If PMC*n* overflows and would signal an exception (PMLCa*n*[CE] and PMGC0[PMIE] are set) while interrupts are disabled (MSR[EE] is clear), and freezing of the counters is not enabled (PMGC0[FCECE] is clear), PMC*n* can wrap around to all zeros again without the performance monitor interrupt being taken.

11.4 Event Counting

This section describes configurability and specific unconditional counting modes.

11.4.1 Processor Context Configurability

Counting can be enabled if conditions in the processor state match a software-specified condition. Because a software task scheduler may switch a processor's execution among multiple processes and because statistics on only a particular process may be of interest, a facility is provided to mark a process. The performance monitor mark bit, MSR[PMM], is used for this purpose. System software may set this bit when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR[PR,PMM] together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified by the PMLCa*n*[FCS,FCU,FCM1,FCM0] fields, the state for which monitoring is enabled, counting is enabled for PMC*n*.

The processor states and the settings of the FCS, FCU, FCM1, and FCM0 fields in PMLCa*n* necessary to enable monitoring of each processor state are shown in Table 11-7.

Table 11-7. Processor States and PMLCa0—PMLCa3 Bit Settings

Processor State	FCS	FCU	FCM1	FCM0
Marked	0	0	0	1
Not marked	0	0	1	0
Supervisor	0	1	0	0
User	1	0	0	0
Marked and supervisor	0	1	0	1
Marked and user	1	0	0	1
Not marked and supervisor	0	1	1	0
Not mark and user	1	0	1	0
All	0	0	0	0
None	X	X	1	1
None	1	1	X	X

Two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR[PMM] and MSR[PR]. This can be accomplished by clearing PMLCa*n*[FCS], PMLCa*n*[FCU], PMLCa*n*[FCM1], and PMLCa*n*[FCM0] for each counter control.
- Counting is unconditionally disabled regardless of the states of MSR[PMM] and MSR[PR]. This can be accomplished by setting PMGC0[FAC] or by setting PMLCa*n*[FC] for each counter control. Alternatively, this can be accomplished by setting PMLCa*n*[FCM1] and PMLCa*n*[FCM0] for each counter control or by setting PMLCa*n*[FCS] and PMLCa*n*[FCU] for each counter control.

11.5 Performance Monitor Application Examples

The following sections provide examples of how to use the performance monitor facility:

11.5.1 Chaining Counters

The counter chaining feature can be used to decrease the processing pollution caused by performance monitor interrupts (things like cache contamination and pipeline effects) by allowing a higher event count than is possible with a single counter. Chaining two counters together effectively adds 32 bits to a counter register where the first counter's overflow event acts like a carry out feeding the second counter. By defining the event of interest to be another PMC's overflow generation, the chained counter increments each time the first counter rolls over to zero. Multiple counters may be chained together.

Because the entire chained value cannot be read in a single instruction, an overflow may occur between counter reads, producing an inaccurate value. A sequence like the following is necessary to read the complete chained value when it spans multiple counters and the counters are not frozen. The example shown is for a two-counter case.

```
loop:   mfpmr           Rx,pmctr1       #load from upper counter
        mfpmr           Ry,pmctr0       #load from lower counter
        mfpmr           Rz,pmctr1       #load from upper counter
        cmp             cr0,0,Rz,Rx     #see if 'old' = 'new'
        bc              4,2,loop        #loop if carry occurred between reads
```

The comparison and loop are necessary to ensure that a consistent set of values has been obtained. The above sequence is not necessary if the counters are frozen.

11.5.2 Event Selection

Event selection is specified through the PMLCa*n* registers described in Section 11.2.3, "Local Control A Registers (PMLCa0–PMLCa3)." The event-select fields in PMLCa*n*[EVENT] are described in Table 11-9, which lists encodings for the selectable events to be monitored. Table 11-9 establishes a correlation between each counter, events to be traced, and the pattern required for the desired selection.

The Spec/Nonspec column indicates whether the event count includes any occurrences due to processing that was not architecturally required by the PowerPC sequential execution model (speculative processing).

- Speculative counts include speculative instructions that were later flushed.
- Nonspeculative counts do not include speculative operations, which are flushed.

Table 11-8 describes how event types are indicated in Table 11-9.

Table 11-8. Event Types

Event Type	Label	Description
Reference	Ref:#	Shared across counters PMC0—PMC3. Applicable to most microprocessors.
Common	Com:#	Shared across counters PMC0–PMC3.
Counter-specific	C[0–3]:#	Counted only on one or more specific counters. The notation indicates the counter to which an event is assigned. For example, an event assigned to counter PMC2 is shown as C2:#.

Table 11-9 describes performance monitor events. Pipeline events in Table 11-9 are defined in Instruction timing.

Table 11-9. Performance Monitor Event Selection

Number	Event	Spec/ Nonspec	Count Description
General Events			
Ref:0	Nothing	Nonspec	Register counter holds current value
Ref:1	Processor cycles	Nonspec	Every processor cycle
Ref:2	Instructions completed	Nonspec	Completed instructions. 0, 1, or 2 per cycle.
Com:4	Instructions fetched	Spec	Fetched instructions. 0, 1, 2, 3, or 4 per cycle. (instructions written to the IQ.)
Com:6	PM_EVENT transitions	Spec	0 to 1 transitions on the *pm_event* input.
Com:7	PM_EVENT cycles	Spec	Processor bus cycles that occur when the *pm_event* input is asserted.
Instruction Types Completed			
Com:8	Branch instructions completed	Nonspec	Completed branch instructions.
Com:9	Load completed	Nonspec	Completed load (**l***, load-update (1 load)), load-multiple (1–32), **dcbt**(L1, CT = 0), and **dcbtst**(L1, CT = 0)
Com:10	Store completed	Nonspec	Completed store (**st***, store-update (1 store)), store-multiple (1–32), **icbi**, **dcbf**, **dcbst**, **dcbt**(CT = 1), **dcbtst**(CT = 1), **dcbz**, **icbt**(CT = 1)
Branch Prediction and Execution Events			
Com:12	Branches finished	Spec	Includes all branch instructions (includes folded branches)
Com:13	Taken branches finished	Spec	Includes all taken branch instructions (includes folded branches)
Com:15	Branches mispredicted (for any reason)	Spec	Counts branch instructions mispredicted due to direction, target (for example if the CTR contents change), or IAB prediction. Does not count instructions that the branch predictor incorrectly predicted to be branches.
Pipeline Stalls			
Com:18	Cycles decode stalled	Spec	Cycles the IQ is not empty but 0 instructions decoded
Com:19	Cycles issue stalled	Spec	Cycles the issue buffer is not empty but 0 instructions issued
Com:31	Cache-inhibited accesses translated	Spec	Cache inhibited accesses translated
Com:61	Number of instruction fetches that hit	Spec	Counts fetches that write at least one instruction to the IQ. (With instruction fetched (com:4), can used to compute instructions-per-fetch)
Instruction MMU and Data MMU Events			
Com:62	MMU inside miss	Spec	Counts instruction TLB miss exceptions
BIU Interface Usage			
Com:67	BIU master requests	Spec	Master transaction starts (assertions of $\overline{ts}$)
Com:68	BIU master instruction-side requests	Spec	Master instruction-side assertions of $\overline{ts}$
Com:69	BIU master data-side requests	Spec	Master data-side assertions of $\overline{ts}$

Table 11-9. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:71	BIU master retries	Spec	Transactions initiated by this processor that were retried on the BIU interface. (The core is master and another device retries the core transaction.)
Snoop			
Com:74	Snoop pushes	N/A	Snoop pushes from all data-side resources. (Counts snoop ARTRYs and WOPs.)
Chaining Events[1]			
Com:82	PMC0 overflow	N/A	PMC0[32] transitions from 1 to 0.
Com:83	PMC1 overflow	N/A	PMC1[32] transitions from 1 to 0.
Com:84	PMC2 overflow	N/A	PMC2[32] transitions from 1 to 0.
Com:85	PMC3 overflow	N/A	PMC3[32] transitioned from 1 to 0.
Interrupt Events			
Com:86	Interrupts taken	Nonspec	—
Com:87	External input interrupts taken	Nonspec	—
Com:88	Critical input interrupts taken	Nonspec	—
Com:89	System call and trap interrupts	Nonspec	—
Ref:90	Transitions of TBL bit selected by PMGC0[TBSEL].	Nonspec	Counts transitions of the TBL bit selected by PMGC0[TBSEL].
e300 Performance Monitor Events			
Com:96	i-cache hits	Spec	Number of fetches that hit in i-cache
Com:97	Instructions folded	Spec	Number of instructions folded (used to determine true number of instructions completed)
Com:100	Stalls due to completion buffer	Spec	Cycles issue stalled due to full completion buffer
Com:101	Reserved	—	Reserved.
Com:104	Stalled completion	Spec	Cycles that completion is stalled
Com:105	Stalles due to load	Spec	Cycles that completion is stalled due to load
Com:106	Stalles due to floating-point	Spec	Cycles that completion is stalled due to floating point instruction
Com:108	Load and stores to cacheable space	Spec	Number of loads and stores to cacheable space in the data cache.
Com:109	Loads and stores that hit in cache	Spec	Number of loads and stores that hit in the data cache.

[1] For chaining events, if a counter is configured to count its own overflow bit, that counter does not increment. For example, if PMC2 is selected to count PMC2 overflow events, PMC2 does not increment.

Appendix A
Instruction Set Listings

This appendix lists the e300 core microprocessor's instruction set as well as the additional PowerPC instructions not implemented in the e300 core. Instructions are sorted by mnemonic, opcode, function, and form. Also included is a quick reference table that contains general information, such as the architecture level, privilege level, and form, and indicates if the instruction is 64-bit and optional.

Note that split fields representing the concatenation of sequences from left to right, are shown in lowercase. For more information refer to Chapter 8, "Instruction Set," in the *Programming Environments Manual*.

The following key applies to the tables in this appendix.

Key: Reserved Bits Instruction Not Implemented in the e300 core

A.1 Instructions Sorted by Mnemonic

Table A-1 lists the instructions implemented in the PowerPC architecture in alphabetical order by mnemonic.

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
add*x*	31		D					A					B					OE	266									Rc
addc*x*	31		D					A					B					OE	10									Rc
adde*x*	31		D					A					B					OE	138									Rc
addi	14		D					A					SIMM															
addic	12		D					A					SIMM															
addic.	13		D					A					SIMM															
addis	15		D					A					SIMM															
addme*x*	31		D					A					0 0 0 0 0					OE	234									Rc
addze*x*	31		D					A					0 0 0 0 0					OE	202									Rc
and*x*	31		S					A					B					28										Rc
andc*x*	31		S					A					B					60										Rc
andi.	28		S					A					UIMM															
andis.	29		S					A					UIMM															
b*x*	18		LI																								AA	LK
bc*x*	16		BO					BI					BD														AA	LK
bcctr*x*	19		BO					BI					0 0 0 0 0					528										LK

Table A-1. Complete Instruction List Sorted by Mnemonic (continued)

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
bclr*x*	19	BO	BI	0 0 0 0 0	16	LK
cmp	31	crfD 0 L	A	B	0	0
cmpi	11	crfD 0 L	A	SIMM		
cmpl	31	crfD 0 L	A	B	32	0
cmpli	10	crfD 0 L	A	UIMM		
cntlzd*x* [1]	31	S	A	0 0 0 0 0	58	Rc
cntlzw*x*	31	S	A	0 0 0 0 0	26	Rc
crand	19	crbD	crbA	crbB	257	0
crandc	19	crbD	crbA	crbB	129	0
creqv	19	crbD	crbA	crbB	289	0
crnand	19	crbD	crbA	crbB	225	0
crnor	19	crbD	crbA	crbB	33	0
cror	19	crbD	crbA	crbB	449	0
crorc	19	crbD	crbA	crbB	417	0
crxor	19	crbD	crbA	crbB	193	0
dcbf	31	0 0 0 0 0	A	B	86	0
dcbi [2]	31	0 0 0 0 0	A	B	470	0
dcbst	31	0 0 0 0 0	A	B	54	0
dcbt	31	0 0 0 0 0	A	B	278	0
dcbtst	31	0 0 0 0 0	A	B	246	0
dcbz	31	0 0 0 0 0	A	B	1014	0
divd*x* [1]	31	D	A	B	OE 489	Rc
divdu*x* [1]	31	D	A	B	OE 457	Rc
divw*x*	31	D	A	B	OE 491	Rc
divwu*x*	31	D	A	B	OE 459	Rc
eciwx [3]	31	D	A	B	310	0
ecowx [3]	31	S	A	B	438	0
eieio	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	854	0
eqv*x*	31	S	A	B	284	Rc
extsb*x*	31	S	A	0 0 0 0 0	954	Rc
extsh*x*	31	S	A	0 0 0 0 0	922	Rc
extsw*x* [1]	31	S	A	0 0 0 0 0	986	Rc
fabs*x*	63	D	0 0 0 0 0	B	264	Rc
fadd*x*	63	D	A	B	0 0 0 0 0 21	Rc
fadds*x*	59	D	A	B	0 0 0 0 0 21	Rc
fcfid*x* [1]	63	D	0 0 0 0 0	B	846	Rc

Table A-1. Complete Instruction List Sorted by Mnemonic (continued)

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fcmpo	63	crfD	0 0	A	B	32		0
fcmpu	63	crfD	0 0	A	B	0		0
fctid*x* [1]	63	D		0 0 0 0 0	B	814		Rc
fctidz*x* [1]	63	D		0 0 0 0 0	B	815		Rc
fctiw*x*	63	D		0 0 0 0 0	B	14		Rc
fctiwz*x*	63	D		0 0 0 0 0	B	15		Rc
fdiv*x*	63	D		A	B	0 0 0 0 0	18	Rc
fdivs*x*	59	D		A	B	0 0 0 0 0	18	Rc
fmadd*x*	63	D		A	B	C	29	Rc
fmadds*x*	59	D		A	B	C	29	Rc
fmr*x*	63	D		0 0 0 0 0	B	72		Rc
fmsub*x*	63	D		A	B	C	28	Rc
fmsubs*x*	59	D		A	B	C	28	Rc
fmul*x*	63	D		A	0 0 0 0 0	C	25	Rc
fmuls*x*	59	D		A	0 0 0 0 0	C	25	Rc
fnabs*x*	63	D		0 0 0 0 0	B	136		Rc
fneg*x*	63	D		0 0 0 0 0	B	40		Rc
fnmadd*x*	63	D		A	B	C	31	Rc
fnmadds*x*	59	D		A	B	C	31	Rc
fnmsub*x*	63	D		A	B	C	30	Rc
fnmsubs*x*	59	D		A	B	C	30	Rc
fres*x* [3]	59	D		0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsp*x*	63	D		0 0 0 0 0	B	12		Rc
frsqrte*x* [3]	63	D		0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsel*x* [3]	63	D		A	B	C	23	Rc
fsqrt*x*	63	D		0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrts*x* [3]	59	D		0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsub*x*	63	D		A	B	0 0 0 0 0	20	Rc
fsubs*x*	59	D		A	B	0 0 0 0 0	20	Rc
icbi	31	0 0 0 0 0		A	B	982		0
icbt [6]	31	0 0 0 0 0		A	B	22		0
isync	19	0 0 0 0 0		0 0 0 0 0	0 0 0 0 0	150		0
lbz	34	D		A	d			
lbzu	35	D		A	d			
lbzux	31	D		A	B	119		0
lbzx	31	D		A	B	87		0

Table A-1. Complete Instruction List Sorted by Mnemonic (continued)

Name	0 5	6 7	8 9	10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29	30 31
ld [1]	58	D			A	ds		0
ldarx [1]	31	D			A	B	84	0
ldu [1]	58	D			A	ds		1
ldux [1]	31	D			A	B	53	0
ldx [1]	31	D			A	B	21	0
lfd	50	D			A	d		
lfdu	51	D			A	d		
lfdux	31	D			A	B	631	0
lfdx	31	D			A	B	599	0
lfs	48	D			A	d		
lfsu	49	D			A	d		
lfsux	31	D			A	B	567	0
lfsx	31	D			A	B	535	0
lha	42	D			A	d		
lhau	43	D			A	d		
lhaux	31	D			A	B	375	0
lhax	31	D			A	B	343	0
lhbrx	31	D			A	B	790	0
lhz	40	D			A	d		
lhzu	41	D			A	d		
lhzux	31	D			A	B	311	0
lhzx	31	D			A	B	279	0
lmw[4]	46	D			A	d		
lswi [4]	31	D			A	NB	597	0
lswx [4]	31	D			A	B	533	0
lwa [1]	58	D			A	ds		2
lwarx	31	D			A	B	20	0
lwaux [1]	31	D			A	B	373	0
lwax [1]	31	D			A	B	341	0
lwbrx	31	D			A	B	534	0
lwz	32	D			A	d		
lwzu	33	D			A	d		
lwzux	31	D			A	B	55	0
lwzx	31	D			A	B	23	0
mcrf	19	crfD	0 0		crfS	0 0	0 0 0 0 0	0	0
mcrfs	63	crfD	0 0		crfS	0 0	0 0 0 0 0	64	0

Table A-1. Complete Instruction List Sorted by Mnemonic (continued)

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mcrxr	31	crfD	0 0	0 0 0 0 0	0 0 0 0 0	512	0
mfcr	31	D		0 0 0 0 0	0 0 0 0 0	19	0
mffs*x*	63	D		0 0 0 0 0	0 0 0 0 0	583	Rc
mfmsr [2]	31	D		0 0 0 0 0	0 0 0 0 0	83	0
mfspr[5]	31	D		spr		339	0
mfsr [2]	31	D		0 SR	0 0 0 0 0	595	0
mfsrin [2]	31	D		0 0 0 0 0	B	659	0
mftb	31	D		tbr		371	0
mtcrf	31	S		0 CRM	0	144	0
mtfsb0*x*	63	crbD		0 0 0 0 0	0 0 0 0 0	70	Rc
mtfsb1*x*	63	crbD		0 0 0 0 0	0 0 0 0 0	38	Rc
mtfsf*x*	63	0 FM		0	B	711	Rc
mtfsfi*x*	63	crfD	0 0	0 0 0 0 0	IMM 0	134	Rc
mtmsr [2]	31	S		0 0 0 0 0	0 0 0 0 0	146	0
mtspr [5]	31	S		spr		467	0
mtsr [2]	31	S		0 SR	0 0 0 0 0	210	0
mtsrin [2]	31	S		0 0 0 0 0	B	242	0
mulhd*x* [1]	31	D		A	B	0 73	Rc
mulhdu*x* [1]	31	D		A	B	0 9	Rc
mulhw*x*	31	D		A	B	0 75	Rc
mulhwu*x*	31	D		A	B	0 11	Rc
mulld*x* [1]	31	D		A	B	OE 233	Rc
mulli	7	D		A	SIMM		
mullw*x*	31	D		A	B	OE 235	Rc
nand*x*	31	S		A	B	476	Rc
neg*x*	31	D		A	0 0 0 0 0	OE 104	Rc
nor*x*	31	S		A	B	124	Rc
or*x*	31	S		A	B	444	Rc
orc*x*	31	S		A	B	412	Rc
ori	24	S		A	UIMM		
oris	25	S		A	UIMM		
rfi [2]	19	0 0 0 0 0		0 0 0 0 0	0 0 0 0 0	50	0
rldcl*x* [1]	30	S		A	B	mb 8	Rc
rldcr*x* [1]	30	S		A	B	me 9	Rc
rldic*x* [1]	30	S		A	sh	mb 2 sh	Rc
rldicl*x* [1]	30	S		A	sh	mb 0 sh	Rc

Table A-1. Complete Instruction List Sorted by Mnemonic (continued)

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26	27 28 29 30	31
rldicr*x* [1]	30	S	A	sh	me	1	sh Rc
rldimi*x* [1]	30	S	A	sh	mb	3	sh Rc
rlwimi*x*	20	S	A	SH	MB	ME	Rc
rlwinm*x*	21	S	A	SH	MB	ME	Rc
rlwnm*x*	23	S	A	B	MB	ME	Rc
sc	17	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			1 0
slbia [1, 2, 3]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	498		0
slbie [1, 2, 3]	31	0 0 0 0 0	0 0 0 0 0	B	434		0
sld*x* [1]	31	S	A	B	27		Rc
slw*x*	31	S	A	B	24		Rc
srad*x* [1]	31	S	A	B	794		Rc
sradi*x* [1]	31	S	A	sh	413		sh Rc
sraw*x*	31	S	A	B	792		Rc
srawi*x*	31	S	A	SH	824		Rc
srd*x* [1]	31	S	A	B	539		Rc
srw*x*	31	S	A	B	536		Rc
stb	38	S	A	d			
stbu	39	S	A	d			
stbux	31	S	A	B	247		0
stbx	31	S	A	B	215		0
std [1]	62	S	A	ds			0
stdcx. [1]	31	S	A	B	214		1
stdu [1]	62	S	A	ds			1
stdux [1]	31	S	A	B	181		0
stdx [1]	31	S	A	B	149		0
stfd	54	S	A	d			
stfdu	55	S	A	d			
stfdux	31	S	A	B	759		0
stfdx	31	S	A	B	727		0
stfiwx [3]	31	S	A	B	983		0
stfs	52	S	A	d			
stfsu	53	S	A	d			
stfsux	31	S	A	B	695		0
stfsx	31	S	A	B	663		0
sth	44	S	A	d			
sthbrx	31	S	A	B	918		0

Table A-1. Complete Instruction List Sorted by Mnemonic (continued)

Name	0 5	6 7	8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
sthu	45	S	A	d			
sthux	31	S	A	B	439		0
sthx	31	S	A	B	407		0
stmw [4]	47	S	A	d			
stswi [4]	31	S	A	NB	725		0
stswx [4]	31	S	A	B	661		0
stw	36	S	A	d			
stwbrx	31	S	A	B	662		0
stwcx.	31	S	A	B	150		1
stwu	37	S	A	d			
stwux	31	S	A	B	183		0
stwx	31	S	A	B	151		0
subf*x*	31	D	A	B	OE	40	Rc
subfc*x*	31	D	A	B	OE	8	Rc
subfe*x*	31	D	A	B	OE	136	Rc
subfic	08	D	A	SIMM			
subfme*x*	31	D	A	0 0 0 0 0	OE	232	Rc
subfze*x*	31	D	A	0 0 0 0 0	OE	200	Rc
sync	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	598		0
td [1]	31	TO	A	B	68		0
tdi [1]	02	TO	A	SIMM			
tlbia [2, 3]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	370		0
tlbie [2, 3]	31	0 0 0 0 0	0 0 0 0 0	B	306		0
tlbld [2, 6]	31	0 0 0 0 0	0 0 0 0 0	B	978		0
tlbli [2, 6]	31	0 0 0 0 0	0 0 0 0 0	B	1010		0
tlbsync [2, 3]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	566		0
tw	31	TO	A	B	4		0
twi	03	TO	A	SIMM			
xor*x*	31	S	A	B	316		Rc
xori	26	S	A	UIMM			
xoris	27	S	A	UIMM			

1 64-bit instruction
2 Supervisor-level instruction
3 Optional in the PowerPC architecture
4 Load and store string or multiple instruction
5 Supervisor- and user-level instruction
6 Implementation-specific instruction

A.2 Instructions Sorted by Opcode

Table A-2 lists the instructions defined in the PowerPC architecture in numeric order by opcode.

Table A-2. Complete Instruction List Sorted by Opcode

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
tdi [1]	0 0 0 0 1 0	TO						A					SIMM															
twi	0 0 0 0 1 1	TO						A					SIMM															
mulli	0 0 0 1 1 1	D						A					SIMM															
subfic	0 0 1 0 0 0	D						A					SIMM															
cmpli	0 0 1 0 1 0	crfD			0		L	A					UIMM															
cmpi	0 0 1 0 1 1	crfD			0		L	A					SIMM															
addic	0 0 1 1 0 0	D						A					SIMM															
addic.	0 0 1 1 0 1	D						A					SIMM															
addi	0 0 1 1 1 0	D						A					SIMM															
addis	0 0 1 1 1 1	D						A					SIMM															
bc*x*	0 1 0 0 0 0	BO						BI					BD													AA	LK	
sc	0 1 0 0 0 1	0 0 0 0 0						0 0 0 0 0					0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0													1	0	
b*x*	0 1 0 0 1 0	LI																								AA	LK	
mcrf	0 1 0 0 1 1	crfD				0 0		crfS				0 0	0 0 0 0 0					0 0 0 0 0 0 0 0 0 0									0	
bclr*x*	0 1 0 0 1 1	BO						BI					0 0 0 0 0					0 0 0 0 0 1 0 0 0 0									LK	
crnor	0 1 0 0 1 1	crbD						crbA					crbB					0 0 0 0 1 0 0 0 0 1									0	
rfi	0 1 0 0 1 1	0 0 0 0 0						0 0 0 0 0					0 0 0 0 0					0 0 0 0 1 1 0 0 1 0									0	
crandc	0 1 0 0 1 1	crbD						crbA					crbB					0 0 1 0 0 0 0 0 0 1									0	
isync	0 1 0 0 1 1	0 0 0 0 0						0 0 0 0 0					0 0 0 0 0					0 0 1 0 0 1 0 1 1 0									0	
crxor	0 1 0 0 1 1	crbD						crbA					crbB					0 0 1 1 0 0 0 0 0 1									0	
crnand	0 1 0 0 1 1	crbD						crbA					crbB					0 0 1 1 1 0 0 0 0 1									0	
crand	0 1 0 0 1 1	crbD						crbA					crbB					0 1 0 0 0 0 0 0 0 1									0	
creqv	0 1 0 0 1 1	crbD						crbA					crbB					0 1 0 0 1 0 0 0 0 1									0	
crorc	0 1 0 0 1 1	crbD						crbA					crbB					0 1 1 0 1 0 0 0 0 1									0	
cror	0 1 0 0 1 1	crbD						crbA					crbB					0 1 1 1 0 0 0 0 0 1									0	
bcctr*x*	0 1 0 0 1 1	BO						BI					0 0 0 0 0					1 0 0 0 0 1 0 0 0 0									LK	
rlwimi*x*	0 1 0 1 0 0	S						A					SH					MB					ME				Rc	
rlwinm*x*	0 1 0 1 0 1	S						A					SH					MB					ME				Rc	
rlwnm*x*	0 1 0 1 1 1	S						A					B					MB					ME				Rc	
ori	0 1 1 0 0 0	S						A					UIMM															
oris	0 1 1 0 0 1	S						A					UIMM															
xori	0 1 1 0 1 0	S						A					UIMM															
xoris	0 1 1 0 1 1	S						A					UIMM															

Table A-2. Complete Instruction List Sorted by Opcode (continued)

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
andi.	011100	S		A	UIMM			
andis.	011101	S		A	UIMM			
rldicl*x* [1]	011110	S		A	sh	mb	000	sh Rc
rldicr*x* [1]	011110	S		A	sh	me	001	sh Rc
rldic*x* [1]	011110	S		A	sh	mb	010	sh Rc
rldimi*x* [1]	011110	S		A	sh	mb	011	sh Rc
rldcl*x* [1]	011110	S		A	B	mb	01000	Rc
rldcr*x* [1]	011110	S		A	B	me	01001	Rc
cmp	011111	crfD	0 L	A	B	0000000000		0
tw	011111	TO		A	B	0000000100		0
subfc*x*	011111	D		A	B	OE 0000001000		Rc
mulhdu*x* [1]	011111	D		A	B	0 0000001001		Rc
addc*x*	011111	D		A	B	OE 0000001010		Rc
mulhwu*x*	011111	D		A	B	0 0000001011		Rc
mfcr	011111	D		00000	00000	0000010011		0
lwarx	011111	D		A	B	0000010100		0
ldx [1]	011111	D		A	B	0000010101		0
lwzx	011111	D		A	B	0000010111		0
slw*x*	011111	S		A	B	0000011000		Rc
cntlzw*x*	011111	S		A	00000	0000011010		Rc
sld*x* [1]	011111	S		A	B	0000011011		Rc
and*x*	011111	S		A	B	0000011100		Rc
cmpl	011111	crfD	0 L	A	B	0000100000		0
subf*x*	011111	D		A	B	OE 0000101000		Rc
ldux [1]	011111	D		A	B	0000110101		0
dcbst	011111	00000		A	B	0000110110		0
lwzux	011111	D		A	B	0000110111		0
cntlzd*x* [1]	011111	S		A	00000	0000111010		Rc
andc*x*	011111	S		A	B	0000111100		Rc
td [1]	011111	TO		A	B	0001000100		0
mulhd*x* [1]	011111	D		A	B	0 0001001001		Rc
mulhw*x*	011111	D		A	B	0 0001001011		Rc
mfmsr	011111	D		00000	00000	0001010011		0
ldarx [1]	011111	D		A	B	0001010100		0
dcbf	011111	00000		A	B	0001010110		0
lbzx	011111	D		A	B	0001010111		0

Table A-2. Complete Instruction List Sorted by Opcode (continued)

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
neg*x*	0 1 1 1 1 1	D	A	0 0 0 0 0	OE 0 0 0 1 1 0 1 0 0 0	Rc
lbzux	0 1 1 1 1 1	D	A	B	0 0 0 1 1 1 0 1 1 1	0
nor*x*	0 1 1 1 1 1	S	A	B	0 0 0 1 1 1 1 1 0 0	Rc
subfe*x*	0 1 1 1 1 1	D	A	B	OE 0 0 1 0 0 0 1 0 0 0	Rc
adde*x*	0 1 1 1 1 1	D	A	B	OE 0 0 1 0 0 0 1 0 1 0	Rc
mtcrf	0 1 1 1 1 1	S	0 CRM	0	0 0 1 0 0 1 0 0 0 0	0
mtmsr	0 1 1 1 1 1	S	0 0 0 0 0	0 0 0 0 0	0 0 1 0 0 1 0 0 1 0	0
stdx [1]	0 1 1 1 1 1	S	A	B	0 0 1 0 0 1 0 1 0 1	0
stwcx.	0 1 1 1 1 1	S	A	B	0 0 1 0 0 1 0 1 1 0	1
stwx	0 1 1 1 1 1	S	A	B	0 0 1 0 0 1 0 1 1 1	0
stdux [1]	0 1 1 1 1 1	S	A	B	0 0 1 0 1 1 0 1 0 1	0
stwux	0 1 1 1 1 1	S	A	B	0 0 1 0 1 1 0 1 1 1	0
subfze*x*	0 1 1 1 1 1	D	A	0 0 0 0 0	OE 0 0 1 1 0 0 1 0 0 0	Rc
addze*x*	0 1 1 1 1 1	D	A	0 0 0 0 0	OE 0 0 1 1 0 0 1 0 1 0	Rc
mtsr	0 1 1 1 1 1	S	0 SR	0 0 0 0 0	0 0 1 1 0 1 0 0 1 0	0
stdcx. [1]	0 1 1 1 1 1	S	A	B	0 0 1 1 0 1 0 1 1 0	1
stbx	0 1 1 1 1 1	S	A	B	0 0 1 1 0 1 0 1 1 1	0
subfme*x*	0 1 1 1 1 1	D	A	0 0 0 0 0	OE 0 0 1 1 1 0 1 0 0 0	Rc
mulld [1]	0 1 1 1 1 1	D	A	B	OE 0 0 1 1 1 0 1 0 0 1	Rc
addme*x*	0 1 1 1 1 1	D	A	0 0 0 0 0	OE 0 0 1 1 1 0 1 0 1 0	Rc
mullw*x*	0 1 1 1 1 1	D	A	B	OE 0 0 1 1 1 0 1 0 1 1	Rc
mtsrin	0 1 1 1 1 1	S	0 0 0 0 0	B	0 0 1 1 1 1 0 0 1 0	0
dcbtst	0 1 1 1 1 1	0 0 0 0 0	A	B	0 0 1 1 1 1 0 1 1 0	0
stbux	0 1 1 1 1 1	S	A	B	0 0 1 1 1 1 0 1 1 1	0
add*x*	0 1 1 1 1 1	D	A	B	OE 0 1 0 0 0 0 1 0 1 0	Rc
dcbt	0 1 1 1 1 1	0 0 0 0 0	A	B	0 1 0 0 0 1 0 1 1 0	0
lhzx	0 1 1 1 1 1	D	A	B	0 1 0 0 0 1 0 1 1 1	0
eqv*x*	0 1 1 1 1 1	S	A	B	0 1 0 0 0 1 1 1 0 0	Rc
tlbie [2, 3]	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	B	0 1 0 0 1 1 0 0 1 0	0
eciwx [3]	0 1 1 1 1 1	D	A	B	0 1 0 0 1 1 0 1 1 0	0
lhzux	0 1 1 1 1 1	D	A	B	0 1 0 0 1 1 0 1 1 1	0
xor*x*	0 1 1 1 1 1	S	A	B	0 1 0 0 1 1 1 1 0 0	Rc
mfspr[4]	0 1 1 1 1 1	D	spr		0 1 0 1 0 1 0 0 1 1	0
lwax [1]	0 1 1 1 1 1	D	A	B	0 1 0 1 0 1 0 1 0 1	0
lhax	0 1 1 1 1 1	D	A	B	0 1 0 1 0 1 0 1 1 1	0
tlbia [2, 3]	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 1 0 1 1 1 0 0 1 0	0

Table A-2. Complete Instruction List Sorted by Opcode (continued)

Name	0	1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
mftb	0 1 1 1 1 1		D	tbr		0 1 0 1 1 1 0 0 1 1		0
lwaux [1]	0 1 1 1 1 1		D	A	B	0 1 0 1 1 1 0 1 0 1		0
lhaux	0 1 1 1 1 1		D	A	B	0 1 0 1 1 1 0 1 1 1		0
sthx	0 1 1 1 1 1		S	A	B	0 1 1 0 0 1 0 1 1 1		0
orc*x*	0 1 1 1 1 1		S	A	B	0 1 1 0 0 1 1 1 0 0		Rc
sradi*x* [1]	0 1 1 1 1 1		S	A	sh	1 1 0 0 1 1 1 0 1 1	sh	Rc
slbie [1, 2, 3]	0 1 1 1 1 1		0 0 0 0 0	0 0 0 0 0	B	0 1 1 0 1 1 0 0 1 0		0
ecowx	0 1 1 1 1 1		S	A	B	0 1 1 0 1 1 0 1 1 0		0
sthux	0 1 1 1 1 1		S	A	B	0 1 1 0 1 1 0 1 1 1		0
or*x*	0 1 1 1 1 1		S	A	B	0 1 1 0 1 1 1 1 0 0		Rc
divdu*x* [1]	0 1 1 1 1 1		D	A	B	OE	0 1 1 1 0 0 1 0 0 1	Rc
divwu*x*	0 1 1 1 1 1		D	A	B	OE	0 1 1 1 0 0 1 0 1 1	Rc
mtspr [4]	0 1 1 1 1 1		S	spr		0 1 1 1 0 1 0 0 1 1		0
dcbi	0 1 1 1 1 1		0 0 0 0 0	A	B	0 1 1 1 0 1 0 1 1 0		0
nand*x*	0 1 1 1 1 1		S	A	B	0 1 1 1 0 1 1 1 0 0		Rc
divd*x* [1]	0 1 1 1 1 1		D	A	B	OE	0 1 1 1 1 0 1 0 0 1	Rc
divw*x*	0 1 1 1 1 1		D	A	B	OE	0 1 1 1 1 0 1 0 1 1	Rc
slbia [1, 2, 3]	0 1 1 1 1 1		0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 1 1 1 1 1 0 0 1 0		0
mcrxr	0 1 1 1 1 1		crfD 0 0	0 0 0 0 0	0 0 0 0 0	1 0 0 0 0 0 0 0 0 0		0
lswx[5]	0 1 1 1 1 1		D	A	B	1 0 0 0 0 1 0 1 0 1		0
lwbrx	0 1 1 1 1 1		D	A	B	1 0 0 0 0 1 0 1 1 0		0
lfsx	0 1 1 1 1 1		D	A	B	1 0 0 0 0 1 0 1 1 1		0
srw*x*	0 1 1 1 1 1		S	A	B	1 0 0 0 0 1 1 0 0 0		Rc
srd*x* [1]	0 1 1 1 1 1		S	A	B	1 0 0 0 0 1 1 0 1 1		Rc
tlbsync [2, 3]	0 1 1 1 1 1		0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	1 0 0 0 1 1 0 1 1 0		0
lfsux	0 1 1 1 1 1		D	A	B	1 0 0 0 1 1 0 1 1 1		0
mfsr	0 1 1 1 1 1		D	0 SR	0 0 0 0 0	1 0 0 1 0 1 0 0 1 1		0
lswi [5]	0 1 1 1 1 1		D	A	NB	1 0 0 1 0 1 0 1 0 1		0
sync	0 1 1 1 1 1		0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	1 0 0 1 0 1 0 1 1 0		0
lfdx	0 1 1 1 1 1		D	A	B	1 0 0 1 0 1 0 1 1 1		0
lfdux	0 1 1 1 1 1		D	A	B	1 0 0 1 1 1 0 1 1 1		0
mfsrin [2]	0 1 1 1 1 1		D	0 0 0 0 0	B	1 0 1 0 0 1 0 0 1 1		0
stswx [5]	0 1 1 1 1 1		S	A	B	1 0 1 0 0 1 0 1 0 1		0
stwbrx	0 1 1 1 1 1		S	A	B	1 0 1 0 0 1 0 1 1 0		0
stfsx	0 1 1 1 1 1		S	A	B	1 0 1 0 0 1 0 1 1 1		0
stfsux	0 1 1 1 1 1		S	A	B	1 0 1 0 1 1 0 1 1 1		0

Table A-2. Complete Instruction List Sorted by Opcode (continued)

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
stswi [5]	011111	S	A	NB	1011010101	0
stfdx	011111	S	A	B	1011010111	0
stfdux	011111	S	A	B	1011111011	0
lhbrx	011111	D	A	B	1100010110	0
sraw*x*	011111	S	A	B	1100011000	Rc
srad*x* [1]	011111	S	A	B	1100011010	Rc
srawi*x*	011111	S	A	SH	1100111000	Rc
eieio	011111	00000	00000	00000	1101010110	0
sthbrx	011111	S	A	B	1110010110	0
extsh*x*	011111	S	A	00000	1110011010	Rc
extsb*x*	011111	S	A	00000	1110111010	Rc
tlbld [2,6]	011111	00000	00000	B	1111010010	0
icbi	011111	00000	A	B	1111010110	0
stfiwx [3]	011111	S	A	B	1111010111	0
extsw [1]	011111	S	A	00000	1111011010	Rc
tlbli [2, 6]	011111	00000	00000	B	1111110010	0
dcbz	011111	00000	A	B	1111110110	0
lwz	100000	D	A	d		
lwzu	100001	D	A	d		
lbz	100010	D	A	d		
lbzu	100011	D	A	d		
stw	100100	S	A	d		
stwu	100101	S	A	d		
stb	100110	S	A	d		
stbu	100111	S	A	d		
lhz	101000	D	A	d		
lhzu	101001	D	A	d		
lha	101010	D	A	d		
lhau	101011	D	A	d		
sth	101100	S	A	d		
sthu	101101	S	A	d		
lmw [5]	101110	D	A	d		
stmw [5]	101111	S	A	d		
lfs	110000	D	A	d		
lfsu	110001	D	A	d		
lfd	110010	D	A	d		

Table A-2. Complete Instruction List Sorted by Opcode (continued)

Name	0 5	6 7 8 9	10 11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
lfdu	1 1 0 0 1 1	D	A	d			
stfs	1 1 0 1 0 0	S	A	d			
stfsu	1 1 0 1 0 1	S	A	d			
stfd	1 1 0 1 1 0	S	A	d			
stfdu	1 1 0 1 1 1	S	A	d			
ld [1]	1 1 1 0 1 0	D	A	ds			0 0
ldu [1]	1 1 1 0 1 0	D	A	ds			0 1
lwa [1]	1 1 1 0 1 0	D	A	ds			1 0
fdivs*x*	1 1 1 0 1 1	D	A	B	0 0 0 0 0	1 0 0 1 0	Rc
fsubs*x*	1 1 1 0 1 1	D	A	B	0 0 0 0 0	1 0 1 0 0	Rc
fadds*x*	1 1 1 0 1 1	D	A	B	0 0 0 0 0	1 0 1 0 1	Rc
fsqrts*x* [3]	1 1 1 0 1 1	D	0 0 0 0 0	B	0 0 0 0 0	1 0 1 1 0	Rc
fres*x* [3]	1 1 1 0 1 1	D	0 0 0 0 0	B	0 0 0 0 0	1 1 0 0 0	Rc
fmuls*x*	1 1 1 0 1 1	D	A	0 0 0 0 0	C	1 1 0 0 1	Rc
fmsubs*x*	1 1 1 0 1 1	D	A	B	C	1 1 1 0 0	Rc
fmadds*x*	1 1 1 0 1 1	D	A	B	C	1 1 1 0 1	Rc
fnmsubs*x*	1 1 1 0 1 1	D	A	B	C	1 1 1 1 0	Rc
fnmadds*x*	1 1 1 0 1 1	D	A	B	C	1 1 1 1 1	Rc
std [1]	1 1 1 1 1 0	S	A	ds			0 0
stdu [1]	1 1 1 1 1 0	S	A	ds			0 1
fcmpu	1 1 1 1 1 1	crfD 0 0	A	B	0 0 0 0 0 0 0 0 0 0		0
frsp*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 0 0 1 1 0 0		Rc
fctiw*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 0 0 1 1 1 0		
fctiwz*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 0 0 1 1 1 1		Rc
fdiv*x*	1 1 1 1 1 1	D	A	B	0 0 0 0 0	1 0 0 1 0	Rc
fsub*x*	1 1 1 1 1 1	D	A	B	0 0 0 0 0	1 0 1 0 0	Rc
fadd*x*	1 1 1 1 1 1	D	A	B	0 0 0 0 0	1 0 1 0 1	Rc
icbt [6]	1 1 1 1 1 1	0 0 0 0 0	A	B	0 0 0 0 0 1 0 1 1 0		0
fsqrt*x* [3]	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 0	1 0 1 1 0	Rc
fsel*x* [3]	1 1 1 1 1 1	D	A	B	C	1 0 1 1 1	Rc
fmul*x*	1 1 1 1 1 1	D	A	0 0 0 0 0	C	1 1 0 0 1	Rc
frsqrte*x* [3]	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 0	1 1 0 1 0	Rc
fmsub*x*	1 1 1 1 1 1	D	A	B	C	1 1 1 0 0	Rc
fmadd*x*	1 1 1 1 1 1	D	A	B	C	1 1 1 0 1	Rc
fnmsub*x*	1 1 1 1 1 1	D	A	B	C	1 1 1 1 0	Rc
fnmadd*x*	1 1 1 1 1 1	D	A	B	C	1 1 1 1 1	Rc

Table A-2. Complete Instruction List Sorted by Opcode (continued)

Name	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fcmpo	1	1	1	1	1	1	crfD			0	0	A					B					0	0	0	0	1	0	0	0	0	0
mtfsb1*x*	1	1	1	1	1	1	crbD					0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0		Rc
fneg*x*	1	1	1	1	1	1	D					0	0	0	0	0	B					0	0	0	0	1	0	1	0	0	0	Rc
mcrfs	1	1	1	1	1	1	crfD			0	0	crfS			0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
mtfsb0*x*	1	1	1	1	1	1	crbD					0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	Rc
fmr*x*	1	1	1	1	1	1	D					0	0	0	0	0	B					0	0	0	1	0	0	1	0	0	0	Rc
mtfsfi*x*	1	1	1	1	1	1	crfD			0	0	0	0	0	0	0	IMM				0	0	0	1	0	0	0	0	1	1	0	Rc
fnabsxv	1	1	1	1	1	1	D					0	0	0	0	0	B					0	0	1	0	0	0	1	0	0	0	Rc
fabs*x*	1	1	1	1	1	1	D					0	0	0	0	0	B					0	1	0	0	0	0	1	0	0	0	Rc
mffs*x*	1	1	1	1	1	1	D					0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	1	1	Rc
mtfsf*x*	1	1	1	1	1	1	0	FM								0	B					1	0	1	1	0	0	0	1	1	1	Rc
fctid*x* [1]	1	1	1	1	1	1	D					0	0	0	0	0	B					1	1	0	0	1	0	1	1	1	0	Rc
fctidz*x* [1]	1	1	1	1	1	1	D					0	0	0	0	0	B					1	1	0	0	1	0	1	1	1	1	Rc
fcfid*x* [1]	1	1	1	1	1	1	D					0	0	0	0	0	B					1	1	0	1	0	0	1	1	1	0	Rc

[1] 64-bit instruction

[2] Supervisor-level instruction

[3] Optional in the PowerPC architecture

[4] Supervisor- and user-level instruction

[5] Load and store string or multiple instruction

[6] e300 core-implementation-specific instruction

A.3 Instructions Grouped by Functional Categories

Table A-3 through Table A-29 list the PowerPC instructions grouped by function.

Table A-3. Integer Arithmetic Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
add*x*	31	D	A	B	OE	266	Rc
addc*x*	31	D	A	B	OE	10	Rc
adde*x*	31	D	A	B	OE	138	Rc
addi	14	D	A	SIMM			
addic	12	D	A	SIMM			
addic.	13	D	A	SIMM			
addis	15	D	A	SIMM			
addme*x*	31	D	A	0 0 0 0 0	OE	234	Rc
addze*x*	31	D	A	0 0 0 0 0	OE	202	Rc
divd*x* [1]	31	D	A	B	OE	489	Rc
divdu*x* [1]	31	D	A	B	OE	457	Rc
divw*x*	31	D	A	B	OE	491	Rc
divwu*x*	31	D	A	B	OE	459	Rc
mulhd*x* [1]	31	D	A	B	0	73	Rc
mulhdu*x* [1]	31	D	A	B	0	9	Rc
mulhw*x*	31	D	A	B	0	75	Rc
mulhwu*x*	31	D	A	B	0	11	Rc
mulld [1]	31	D	A	B	OE	233	Rc
mulli	07	D	A	SIMM			
mullw*x*	31	D	A	B	OE	235	Rc
neg*x*	31	D	A	0 0 0 0 0	OE	104	Rc
subf*x*	31	D	A	B	OE	40	Rc
subfc*x*	31	D	A	B	OE	8	Rc
subfic*x*	08	D	A	SIMM			
subfe*x*	31	D	A	B	OE	136	Rc
subfme*x*	31	D	A	0 0 0 0 0	OE	232	Rc
subfze*x*	31	D	A	0 0 0 0 0	OE	200	Rc

[1] 64-bit instruction

Table A-4. Integer Compare Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
cmp	31		crfD			0	L	A					B					0 0 0 0 0 0 0 0 0 0										0
cmpi	11		crfD			0	L	A					SIMM															
cmpl	31		crfD			0	L	A					B					32										0
cmpli	10		crfD			0	L	A					UIMM															

Table A-5. Integer Logical Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
and*x*	31		S					A					B					28										Rc
andc*x*	31		S					A					B					60										Rc
andi.	28		S					A					UIMM															
andis.	29		S					A					UIMM															
cntlzd*x* [1]	31		S					A					0 0 0 0 0					58										Rc
cntlzw*x*	31		S					A					0 0 0 0 0					26										Rc
eqv*x*	31		S					A					B					284										Rc
extsb*x*	31		S					A					0 0 0 0 0					954										Rc
extsh*x*	31		S					A					0 0 0 0 0					922										Rc
extsw*x* [1]	31		S					A					0 0 0 0 0					986										Rc
nand*x*	31		S					A					B					476										Rc
nor*x*	31		S					A					B					124										Rc
or*x*	31		S					A					B					444										Rc
orc*x*	31		S					A					B					412										Rc
ori	24		S					A					UIMM															
oris	25		S					A					UIMM															
xor*x*	31		S					A					B					316										Rc
xori	26		S					A					UIMM															
xoris	27		S					A					UIMM															

[1] 64-bit instruction

Table A-6. Integer Rotate Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rldcl*x* [1]	30		S					A					B					mb						8				Rc
rldcr*x* [1]	30		S					A					B					me						9				Rc
rldic*x* [1]	30		S					A					sh					mb						2		sh		Rc
rldicl*x* [1]	30		S					A					sh					mb						0		sh		Rc
rldicr*x* [1]	30		S					A					sh					me						1		sh		Rc

Table A-6. Integer Rotate Instructions (continued)

rldimi*x* [1]	30	S	A	sh	mb	3	sh	Rc
rlwimi*x*	22	S	A	SH	MB	ME		Rc
rlwinm*x*	20	S	A	SH	MB	ME		Rc
rlwnm*x*	21	S	A	SH	MB	ME		Rc

[1] 64-bit instruction

Table A-7. Integer Shift Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
sld*x* [1]	31	S	A	B	27	Rc
slw*x*	31	S	A	B	24	Rc
srad*x* [1]	31	S	A	B	794	Rc
sradi*x* [1]	31	S	A	sh	413 sh	Rc
sraw*x*	31	S	A	B	792	Rc
srawi*x*	31	S	A	SH	824	Rc
srd*x* [1]	31	S	A	B	539	Rc
srw*x*	31	S	A	B	536	Rc

[1] 64-bit instruction

Table A-8. Floating-Point Arithmetic Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fadd*x*	63	D	A	B	0 0 0 0 0	21	Rc
fadds*x*	59	D	A	B	0 0 0 0 0	21	Rc
fdiv*x*	63	D	A	B	0 0 0 0 0	18	Rc
fdivs*x*	59	D	A	B	0 0 0 0 0	18	Rc
fmul*x*	63	D	A	0 0 0 0 0	C	25	Rc
fmuls*x*	59	D	A	0 0 0 0 0	C	25	Rc
fres*x* [1]	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsqrte*x* [1]	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsub*x*	63	D	A	B	0 0 0 0 0	20	Rc
fsubs*x*	59	D	A	B	0 0 0 0 0	20	Rc
fsel*x* [1]	63	D	A	B	C	23	Rc
fsqrt*x* [1]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrts*x* [1]	59	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc

[1] Optional in the PowerPC architecture

Table A-9. Floating-Point Multiply-Add Instructions

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
fmadd*x*	63	D		A	B	C	29	Rc
fmadds*x*	59	D		A	B	C	29	Rc
fmsub*x*	63	D		A	B	C	28	Rc
fmsubs*x*	59	D		A	B	C	28	Rc
fnmadd*x*	63	D		A	B	C	31	Rc
fnmadds*x*	59	D		A	B	C	31	Rc
fnmsub*x*	63	D		A	B	C	30	Rc
fnmsubs*x*	59	D		A	B	C	30	Rc

Table A-10. Floating-Point Rounding and Conversion Instructions

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
fcfid*x* [1]	63	D		0 0 0 0 0	B	846	Rc
fctid*x* [1]	63	D		0 0 0 0 0	B	814	Rc
fctidz*x* [1]	63	D		0 0 0 0 0	B	815	Rc
fctiw*x*	63	D		0 0 0 0 0	B	14	Rc
fctiwz*x*	63	D		0 0 0 0 0	B	15	Rc
frsp*x*	63	D		0 0 0 0 0	B	12	Rc

[1] 64-bit instruction

Table A-11. Floating-Point Compare Instructions

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
fcmpo	63	crfD	0 0	A	B	32	0
fcmpu	63	crfD	0 0	A	B	0	0

Table A-12. Floating-Point Status and Control Register Instructions

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mcrfs	63	crfD	0 0	crfS 0 0	0 0 0 0 0	64	0
mffs*x*	63	D		0 0 0 0 0	0 0 0 0 0	583	Rc
mtfsb0*x*	63	crbD		0 0 0 0 0	0 0 0 0 0	70	Rc
mtfsb1*x*	63	crbD		0 0 0 0 0	0 0 0 0 0	38	Rc
mtfsf*x*	31	0	FM	0	B	711	Rc
mtfsfi*x*	63	crfD	0 0	0 0 0 0 0	IMM 0	134	Rc

Table A-13. Integer Load Instructions

Name	0 5	6 7	8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lbz	34	D	A	d			
lbzu	35	D	A	d			
lbzux	31	D	A	B		119	0
lbzx	31	D	A	B		87	0
ld [1]	58	D	A	ds			0
ldu [1]	58	D	A	ds			1
ldux [1]	31	D	A	B		53	0
ldx [1]	31	D	A	B		21	0
lha	42	D	A	d			
lhau	43	D	A	d			
lhaux	31	D	A	B		375	0
lhax	31	D	A	B		343	0
lhz	40	D	A	d			
lhzu	41	D	A	d			
lhzux	31	D	A	B		311	0
lhzx	31	D	A	B		279	0
lwa [1]	58	D	A	ds			2
lwaux [1]	31	D	A	B		373	0
lwax [1]	31	D	A	B		341	0
lwz	32	D	A	d			
lwzu	33	D	A	d			
lwzux	31	D	A	B		55	0
lwzx	31	D	A	B		23	0

[1] 64-bit instruction

Table A-14. Integer Store Instructions

Name	0 5	6 7	8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
stb	38	S	A	d			
stbu	39	S	A	d			
stbux	31	S	A	B		247	0
stbx	31	S	A	B		215	0
std [1]	62	S	A	ds			0
stdu [1]	62	S	A	ds			1
stdux [1]	31	S	A	B		181	0
stdx [1]	31	S	A	B		149	0

Table A-14. Integer Store Instructions (continued)

sth	44	S	A	d		
sthu	45	S	A	d		
sthux	31	S	A	B	439	0
sthx	31	S	A	B	407	0
stw	36	S	A	d		
stwu	37	S	A	d		
stwux	31	S	A	B	183	0
stwx	31	S	A	B	151	0

[1] 64-bit instruction

Table A-15. Integer Load and Store with Byte-Reverse Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lhbrx	31	D						A					B					790										0
lwbrx	31	D						A					B					534										0
sthbrx	31	S						A					B					918										0
stwbrx	31	S						A					B					662										0

Table A-16. Integer Load and Store Multiple Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lmw [1]	46	D						A					d															
stmw [1]	47	S						A					d															

[1] Load and store string or multiple instruction

Table A-17. Integer Load and Store String Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lswi [1]	31	D						A					NB					597										0
lswx [1]	31	D						A					B					533										0
stswi [1]	31	S						A					NB					725										0
stswx [1]	31	S						A					B					661										0

[1] Load and store string or multiple instruction

Table A-18. Memory Synchronization Instructions

Name	0	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
eieio	31		0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	854	0
isync	19		0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150	0
ldarx [1]	31		D	A	B	84	0
lwarx	31		D	A	B	20	0
stdcx [1]	31		S	A	B	214	1
stwcx.	31		S	A	B	150	1
sync	31		0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	598	0

[1] 64-bit instruction

Table A-19. Floating-Point Load Instructions

Name	0	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lfd	50		D	A	d		
lfdu	51		D	A	d		
lfdux	31		D	A	B	631	0
lfdx	31		D	A	B	599	0
lfs	48		D	A	d		
lfsu	49		D	A	d		
lfsux	31		D	A	B	567	0
lfsx	31		D	A	B	535	0

Table A-20. Floating-Point Store Instructions

Name	0	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
stfd	54		S	A	d		
stfdu	55		S	A	d		
stfdux	31		S	A	B	759	0
stfdx	31		S	A	B	727	0
stfiwx [1]	31		S	A	B	983	0
stfs	52		S	A	d		
stfsu	53		S	A	d		
stfsux	31		S	A	B	695	0
stfsx	31		S	A	B	663	0

[1] Optional in the PowerPC architecture

Table A-21. Floating-Point Move Instructions

Name	0	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
fabs*x*	63	D	0 0 0 0 0	B	264		Rc
fmr*x*	63	D	0 0 0 0 0	B	72		Rc
fnabs*x*	63	D	0 0 0 0 0	B	136		Rc
fneg*x*	63	D	0 0 0 0 0	B	40		Rc

Table A-22. Branch Instructions

Name	0	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
b*x*	18	LI					AA LK
bc*x*	16	BO		BI	BD		AA LK
bcctr*x*	19	BO		BI	0 0 0 0 0	528	LK
bclr*x*	19	BO		BI	0 0 0 0 0	16	LK

Table A-23. Condition Register Logical Instructions

Name	0	5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
crand	19	crbD			crbA	crbB	257	0
crandc	19	crbD			crbA	crbB	129	0
creqv	19	crbD			crbA	crbB	289	0
crnand	19	crbD			crbA	crbB	225	0
crnor	19	crbD			crbA	crbB	33	0
cror	19	crbD			crbA	crbB	449	0
crorc	19	crbD			crbA	crbB	417	0
crxor	19	crbD			crbA	crbB	193	0
mcrf	19	crfD		0 0	crfS	0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0	0

Table A-24. System Linkage Instructions

Name	0	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
rfi [1]	19	0 0 0 0 0		0 0 0 0 0	0 0 0 0 0	50	0
sc	17	0 0 0 0 0		0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1	0

[1] Supervisor-level instruction

Table A-25. Trap Instructions

Name	0	5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
td [1]	31		TO		A	B	68	0
tdi [1]	03		TO		A	SIMM		
tw	31		TO		A	B	4	0
twi	03		TO		A	SIMM		

[1] 64-bit instruction

Table A-26. Processor Control Instructions

Name	0	5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mcrxr	31		crfS	0 0	0 0 0 0 0	0 0 0 0 0	512	0
mfcr	31		D		0 0 0 0 0	0 0 0 0 0	19	0
mfmsr [1]	31		D		0 0 0 0 0	0 0 0 0 0	83	0
mfspr [2]	31		D		spr		339	0
mftb	31		D		tpr		371	0
mtcrf	31		S		0	CRM	0	144	0
mtmsr [1]	31		S		0 0 0 0 0	0 0 0 0 0	146	0
mtspr [2]	31		D		spr		467	0

[1] Supervisor-level instruction

[2] Supervisor- and user-level instruction

Table A-27. Cache Management Instructions

Name	0	5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
dcbf	31		0 0 0 0 0		A	B	86	0
dcbi [1]	31		0 0 0 0 0		A	B	470	0
dcbst	31		0 0 0 0 0		A	B	54	0
dcbt	31		0 0 0 0 0		A	B	278	0
dcbtst	31		0 0 0 0 0		A	B	246	0
dcbz	31		0 0 0 0 0		A	B	1014	0
icbi	31		0 0 0 0 0		A	B	982	0
icbt[2]	31		0 0 0 0 0		A	B	22	0

[1] Supervisor-level instruction

[2] e300 core-implementation-specific instruction

Table A-28. Segment Register Manipulation Instructions

Name	0 5	6 7 8 9 10	11	12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mfsr [1]	31	D	0	SR	0 0 0 0 0	595	0
mfsrin [1]	31	D	0 0 0 0 0		B	659	0
mtsr [1]	31	S	0	SR	0 0 0 0 0	210	0
mtsrin [1]	31	S	0 0 0 0 0		B	242	0

[1] Supervisor-level instruction

Table A-29. Lookaside Buffer Management Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
slbia [1, 2, 3]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	498	0
slbie [1, 2, 3]	31	0 0 0 0 0	0 0 0 0 0	B	434	0
tlbia [1, 3]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	370	0
tlbie [1, 3]	31	0 0 0 0 0	0 0 0 0 0	B	306	0
tlbld [1,4]	31	0 0 0 0 0	0 0 0 0 0	B	978	0
tlbli [1, 4]	31	0 0 0 0 0	0 0 0 0 0	B	1010	0
tlbsync [1, 3]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	566	0

[1] Supervisor-level instruction
[2] 64-bit instruction
[3] Optional in the PowerPC architecture
[4] e300 core-implementation specific instruction

A.4 Instructions Sorted by Form

Table A-30 through Table A-44 list the PowerPC instructions grouped by form.

Table A-30. I-Form

Name	0 5	6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	30	31
	OPCD	LI	AA	LK
Specific Instruction				
b*x*	18	LI	AA	LK

Table A-31. B-Form

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29	30	31
	OPCD	BO	BI	BD	AA	LK
Specific Instruction						
bc*x*	16	BO	BI	BD	AA	LK

Table A-32. SC-Form

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29	30	31
	OPCD	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1	0
Specific Instruction						
sc	17	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1	0

Table A-33. D-Form

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
	OPCD	D	A	d
	OPCD	D	A	SIMM
	OPCD	S	A	d
	OPCD	S	A	UIMM
	OPCD	crfD 0 L	A	SIMM
	OPCD	crfD 0 L	A	UIMM
	OPCD	TO	A	SIMM
Specific Instruction				
addi	14	D	A	SIMM
addic	12	D	A	SIMM
addic.	13	D	A	SIMM
addis	15	D	A	SIMM
andi.	28	S	A	UIMM
andis.	29	S	A	UIMM
cmpi	11	crfD 0 L	A	SIMM
cmpli	10	crfD 0 L	A	UIMM
lbz	34	D	A	d
lbzu	35	D	A	d
lfd	50	D	A	d
lfdu	51	D	A	d
lfs	48	D	A	d
lfsu	49	D	A	d
lha	42	D	A	d
lhau	43	D	A	d
lhz	40	D	A	d
lhzu	41	D	A	d
lmw [1]	46	D	A	d
lwz	32	D	A	d

Table A-33. D-Form (continued)

lwzu	33	D	A	d
mulli	7	D	A	SIMM
ori	24	S	A	UIMM
oris	25	S	A	UIMM
stb	38	S	A	d
stbu	39	S	A	d
stfd	54	S	A	d
stfdu	55	S	A	d
stfs	52	S	A	d
stfsu	53	S	A	d
sth	44	S	A	d
sthu	45	S	A	d
stmw [1]	47	S	A	d
stw	36	S	A	d
stwu	37	S	A	d

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

subfic	08	D	A	SIMM
tdi [2]	02	TO	A	SIMM
twi	03	TO	A	SIMM
xori	26	S	A	UIMM
xoris	27	S	A	UIMM

[1] Load and store string or multiple instruction

[2] 64-bit instruction

Table A-34. DS-Form

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

	OPCD	D	A	ds	XO
	OPCD	S	A	ds	XO

Specific Instructions

ld [1]	58	D	A	ds	0
ldu [1]	58	D	A	ds	1
lwa [1]	58	D	A	ds	2
std [1]	62	S	A	ds	0
stdu [1]	62	S	A	ds	1

[1] 64-bit instruction

Table A-35. X-Form

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
	OPCD	D		A	B	XO	0
	OPCD	D		A	NB	XO	0
	OPCD	D		0 0 0 0 0	B	XO	0
	OPCD	D		0 0 0 0 0	0 0 0 0 0	XO	0
	OPCD	D		0 SR	0 0 0 0 0	XO	0
	OPCD	S		A	B	XO	Rc
	OPCD	S		A	B	XO	1
	OPCD	S		A	B	XO	0
	OPCD	S		A	NB	XO	0
	OPCD	S		A	0 0 0 0 0	XO	Rc
	OPCD	S		0 0 0 0 0	B	XO	0
	OPCD	S		0 0 0 0 0	0 0 0 0 0	XO	0
	OPCD	S		0 SR	0 0 0 0 0	XO	0
	OPCD	S		A	SH	XO	Rc
	OPCD	crfD	0 L	A	B	XO	0
	OPCD	crfD	0 0	A	B	XO	0
	OPCD	crfD	0 0	crfS 0 0	0 0 0 0 0	XO	0
	OPCD	crfD	0 0	0 0 0 0 0	0 0 0 0 0	XO	0
	OPCD	crfD	0 0	0 0 0 0 0	IMM 0	XO	Rc
	OPCD	TO		A	B	XO	0
	OPCD	D		0 0 0 0 0	B	XO	Rc
	OPCD	D		0 0 0 0 0	0 0 0 0 0	XO	Rc
	OPCD	crbD		0 0 0 0 0	0 0 0 0 0	XO	Rc
	OPCD	0 0 0 0 0		A	B	XO	0
	OPCD	0 0 0 0 0		0 0 0 0 0	B	XO	0
	OPCD	0 0 0 0 0		0 0 0 0 0	0 0 0 0 0	XO	0

Specific Instructions

and*x*	31	S		A	B	28	Rc
andc*x*	31	S		A	B	60	Rc
cmp	31	crfD	0 L	A	B	0	0
cmpl	31	crfD	0 L	A	B	32	0
cntlzd*x* [1]	31	S		A	0 0 0 0 0	58	Rc
cntlzw*x*	31	S		A	0 0 0 0 0	26	Rc
dcbf	31	0 0 0 0 0		A	B	86	0

Table A-35. X-Form (continued)

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
dcbi [2]	31	0 0 0 0 0		A	B	470	0
dcbst	31	0 0 0 0 0		A	B	54	0
dcbt	31	0 0 0 0 0		A	B	278	0
dcbtst	31	0 0 0 0 0		A	B	246	0
dcbz	31	0 0 0 0 0		A	B	1014	0
eciwx [3]	31	D		A	B	310	0
ecowx [3]	31	S		A	B	438	0
eieio	31	0 0 0 0 0		0 0 0 0 0	0 0 0 0 0	854	0
eqv*x*	31	S		A	B	284	Rc
extsb*x*	31	S		A	0 0 0 0 0	954	Rc
extsh*x*	31	S		A	0 0 0 0 0	922	Rc
extsw*x* [1]	31	S		A	0 0 0 0 0	986	Rc
fabs*x*	63	D		0 0 0 0 0	B	264	Rc
fcfid*x* [1]	63	D		0 0 0 0 0	B	846	Rc
fcmpo	63	crfD	0 0	A	B	32	0
fcmpu	63	crfD	0 0	A	B	0	0
fctid*x* [1]	63	D		0 0 0 0 0	B	814	Rc
fctidz*x* [1]	63	D		0 0 0 0 0	B	815	Rc
fctiw*x*	63	D		0 0 0 0 0	B	14	Rc
fctiwz*x*	63	D		0 0 0 0 0	B	15	Rc
fmr*x*	63	D		0 0 0 0 0	B	72	Rc
fnabs*x*	63	D		0 0 0 0 0	B	136	Rc
fneg*x*	63	D		0 0 0 0 0	B	40	Rc
frsp*x*	63	D		0 0 0 0 0	B	12	Rc
icbi	31	0 0 0 0 0		A	B	982	0
icbt [5]	31	0 0 0 0 0		A	B	22	0
lbzux	31	D		A	B	119	0
lbzx	31	D		A	B	87	0
ldarx [1]	31	D		A	B	84	0
ldux [1]	31	D		A	B	53	0
ldx [1]	31	D		A	B	21	0
lfdux	31	D		A	B	631	0
lfdx	31	D		A	B	599	0
lfsux	31	D		A	B	567	0
lfsx	31	D		A	B	535	0

Table A-35. X-Form (continued)

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lhaux	31	D	A	B	375	0
lhax	31	D	A	B	343	0
lhbrx	31	D	A	B	790	0
lhzux	31	D	A	B	311	0
lhzx	31	D	A	B	279	0
lswi [3]	31	D	A	NB	597	0
lswx [3]	31	D	A	B	533	0
lwarx	31	D	A	B	20	0
lwaux [1]	31	D	A	B	373	0
lwax [1]	31	D	A	B	341	0
lwbrx	31	D	A	B	534	0
lwzux	31	D	A	B	55	0
lwzx	31	D	A	B	23	0
mcrfs	63	crfD 0 0	crfS 0 0	0 0 0 0 0	64	0
mcrxr	31	crfD 0 0	0 0 0 0 0	0 0 0 0 0	512	0
mfcr	31	D	0 0 0 0 0	0 0 0 0 0	19	0
mffs*x*	63	D	0 0 0 0 0	0 0 0 0 0	583	Rc
mfmsr [2]	31	D	0 0 0 0 0	0 0 0 0 0	83	0
mfsr [2]	31	D	0 SR	0 0 0 0 0	595	0
mfsrin [2]	31	D	0 0 0 0 0	B	659	0
mtfsb0*x*	63	crbD	0 0 0 0 0	0 0 0 0 0	70	Rc
mtfsb1*x*	63	crfD	0 0 0 0 0	0 0 0 0 0	38	Rc
mtfsfi*x*	63	crbD 0 0	0 0 0 0 0	IMM 0	134	Rc
mtmsr [2]	31	S	0 0 0 0 0	0 0 0 0 0	146	0
mtsr [2]	31	S	0 SR	0 0 0 0 0	210	0
mtsrin [2]	31	S	0 0 0 0 0	B	242	0
nand*x*	31	S	A	B	476	Rc
nor*x*	31	S	A	B	124	Rc
or*x*	31	S	A	B	444	Rc
orc*x*	31	S	A	B	412	Rc
slbia [1, 2, 4]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	498	0
slbie [1, 2, 4]	31	0 0 0 0 0	0 0 0 0 0	B	434	0
sld*x* [1]	31	S	A	B	27	Rc
slw*x*	31	S	A	B	24	Rc
srad*x* [1]	31	S	A	B	794	Rc

Table A-35. X-Form (continued)

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
sraw*x*	31	S	A	B	792	Rc
srawi*x*	31	S	A	SH	824	Rc
srd*x* [1]	31	S	A	B	539	Rc
srw*x*	31	S	A	B	536	Rc
stbux	31	S	A	B	247	0
stbx	31	S	A	B	215	0
stdcx. [1]	31	S	A	B	214	1
stdux [1]	31	S	A	B	181	0
stdx [1]	31	S	A	B	149	0
stfdux	31	S	A	B	759	0
stfdx	31	S	A	B	727	0
stfiwx [4]	31	S	A	B	983	0
stfsux	31	S	A	B	695	0
stfsx	31	S	A	B	663	0
sthbrx	31	S	A	B	918	0
sthux	31	S	A	B	439	0
sthx	31	S	A	B	407	0
stswi [3]	31	S	A	NB	725	0
stswx [3]	31	S	A	B	661	0
stwbrx	31	S	A	B	662	0
stwcx.	31	S	A	B	150	1
stwux	31	S	A	B	183	0
stwx	31	S	A	B	151	0
sync	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	598	0
td [1]	31	TO	A	B	68	0
tlbia [2, 4]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	370	0
tlbie [2, 4]	31	0 0 0 0 0	0 0 0 0 0	B	306	0
tlbld [2, 5]	31	0 0 0 0 0	0 0 0 0 0	B	978	0
tlbli [2, 5]	31	0 0 0 0 0	0 0 0 0 0	B	1010	0
tlbsync [2, 4]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	566	0
tw	31	TO	A	B	4	0
xor*x*	31	S	A	B	316	Rc

[1] 64-bit instruction

[2] Supervisor- and user-level instruction

[3] Load and store string or multiple instruction

[4] Optional in the PowerPC architecture

[5] e300 core-implementation specific instruction

Table A-36. XL-Form

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
	OPCD	BO	BI	0 0 0 0 0	XO	LK
	OPCD	crbD	crbA	crbB	XO	0
	OPCD	crfD 0 0	crfS 0 0	0 0 0 0 0	XO	0
	OPCD	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	XO	0
Specific Instructions						
bcctr*x*	19	BO	BI	0 0 0 0 0	528	LK
bclr*x*	19	BO	BI	0 0 0 0 0	16	LK
crand	19	crbD	crbA	crbB	257	0
crandc	19	crbD	crbA	crbB	129	0
creqv	19	crbD	crbA	crbB	289	0
crnand	19	crbD	crbA	crbB	225	0
crnor	19	crbD	crbA	crbB	33	0
cror	19	crbD	crbA	crbB	449	0
crorc	19	crbD	crbA	crbB	417	0
crxor	19	crbD	crbA	crbB	193	0
isync	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150	0
mcrf	19	crfD 0 0	crfS 0 0	0 0 0 0 0	0	0
rfi [1]	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	50	0

[1] Supervisor-level instruction

Table A-37. XFX-Form

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
	OPCD	D	spr		XO	0
	OPCD	D	0	CRM	0 XO	0
	OPCD	S	spr		XO	0
	OPCD	D	tbr		XO	0
Specific Instructions						
mfspr [1]	31	D	spr		339	0
mftb	31	D	tbr		371	0
mtcrf	31	S	0	CRM	0 144	0
mtspr [1]	31	D	spr		467	0

[1] Supervisor- and user-level instruction

Table A-38. XFL-Form

Name	0 5	6	7 8 9 10	11	12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
	OPCD	0	FM	0	B	XO		Rc
Specific Instructions								
mtfsf*x*	63	0	FM	0	B	711		Rc

Table A-39. XS-Form

Name	0 5	6	7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
	OPCD	S		A	sh	XO	sh Rc
Specific Instructions							
sradi*x* [1]	31	S		A	sh	413	sh Rc

[1] 64-bit instruction

Table A-40. XO-Form

Name	0 5	6	7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
	OPCD	D		A	B	OE	XO	Rc
	OPCD	D		A	B	0	XO	Rc
	OPCD	D		A	0 0 0 0 0	OE	XO	Rc
Specific Instructions								
add*x*	31	D		A	B	OE	266	Rc
addc*x*	31	D		A	B	OE	10	Rc
adde*x*	31	D		A	B	OE	138	Rc
addme*x*	31	D		A	0 0 0 0 0	OE	234	Rc
addze*x*	31	D		A	0 0 0 0 0	OE	202	Rc
divd*x* [1]	31	D		A	B	OE	489	Rc
divdu*x* [1]	31	D		A	B	OE	457	Rc
divw*x*	31	D		A	B	OE	491	Rc
divwu*x*	31	D		A	B	OE	459	Rc
mulhd*x* [1]	31	D		A	B	0	73	Rc
mulhdu*x* [1]	31	D		A	B	0	9	Rc
mulhw*x*	31	D		A	B	0	75	Rc
mulhwu*x*	31	D		A	B	0	11	Rc
mulld*x* [1]	31	D		A	B	OE	233	Rc
mullw*x*	31	D		A	B	OE	235	Rc
neg*x*	31	D		A	0 0 0 0 0	OE	104	Rc
subf*x*	31	D		A	B	OE	40	Rc

Table A-40. XO-Form (continued)

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
subfc*x*	31	D	A	B	OE	8	Rc
subfe*x*	31	D	A	B	OE	136	Rc
subfme*x*	31	D	A	0 0 0 0 0	OE	232	Rc
subfze*x*	31	D	A	0 0 0 0 0	OE	200	Rc

[1] 64-bit instruction

Table A-41. A-Form

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
	OPCD	D	A	B	0 0 0 0 0	XO	Rc
	OPCD	D	A	B	C	XO	Rc
	OPCD	D	A	0 0 0 0 0	C	XO	Rc
	OPCD	D	0 0 0 0 0	B	0 0 0 0 0	XO	Rc
Specific Instructions							
fadd*x*	63	D	A	B	0 0 0 0 0	21	Rc
fadds*x*	59	D	A	B	0 0 0 0 0	21	Rc
fdiv*x*	63	D	A	B	0 0 0 0 0	18	Rc
fdivs*x*	59	D	A	B	0 0 0 0 0	18	Rc
fmadd*x*	63	D	A	B	C	29	Rc
fmadds*x*	59	D	A	B	C	29	Rc
fmsub*x*	63	D	A	B	C	28	Rc
fmsubs*x*	59	D	A	B	C	28	Rc
fmul*x*	63	D	A	0 0 0 0 0	C	25	Rc
fmuls*x*	59	D	A	0 0 0 0 0	C	25	Rc
fnmadd*x*	63	D	A	B	C	31	Rc
fnmadds*x*	59	D	A	B	C	31	Rc
fnmsub*x*	63	D	A	B	C	30	Rc
fnmsubs*x*	59	D	A	B	C	30	Rc
fres*x* [1]	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsqrte*x* [1]	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsel*x* [1]	63	D	A	B	C	23	Rc
fsqrt*x* [1]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrts*x* [1]	59	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsub*x*	63	D	A	B	0 0 0 0 0	20	Rc
fsubs*x*	59	D	A	B	0 0 0 0 0	20	Rc

[1] Optional in the PowerPC architecture

Table A-42. M-Form

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	OPCD				S				A					SH					MB						ME			Rc
	OPCD				S				A					B					MB						ME			Rc
Specific Instructions																												
rlwimi*x*	20				S				A					SH					MB						ME			Rc
rlwinm*x*	21				S				A					SH					MB						ME			Rc
rlwnm*x*	23				S				A					B					MB						ME			Rc

Table A-43. MD-Form

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	OPCD				S				A					sh					mb						XO		sh	Rc
	OPCD				S				A					sh					me						XO		sh	Rc
Specific Instructions																												
ridicx [1]	30				S				A					sh					mb						2		sh	Rc
rldicl*x* [1]	30				S				A					sh					mb						0		sh	Rc
rldicr*x* [1]	30				S				A					sh					me						1		sh	Rc
rldimi*x* [1]	30				S				A					sh					mb						3		sh	Rc

[1] 64-bit instruction

Table A-44. MDS-Form

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	OPCD				S				A					B					mb						XO			Rc
	OPCD				S				A					B					me						XO			Rc
Specific Instructions																												
rldcl*x* [1]	30				S				A					B					mb						8			Rc
rldcr*x* [1]	30				S				A					B					me						9			Rc

[1] 64-bit instruction

A.5 Instruction Set Legend

Table A-45 provides general information on the PowerPC instruction set (such as the architectural level, privilege level, and form).

Table A-45. PowerPC Instruction Set Legend

	UISA	VEA	OEA	Supervisor Level	Optional	64-Bit	Form
add*x*	√						XO
addc*x*	√						XO
adde*x*	√						XO
addi	√						D
addic	√						D
addic.	√						D
addis	√						D
addme*x*	√						XO
addze*x*	√						XO
and*x*	√						X
andc*x*	√						X
andi.	√						D
andis.	√						D
b*x*	√						I
bc*x*	√						B
bcctr*x*	√						XL
bclr*x*	√						XL
cmp	√						X
cmpi	√						D
cmpl	√						X
cmpli	√						D
cntlzd*x* [1]	√				√		X
cntlzw*x*	√						X
crand	√						XL
crandc	√						XL
creqv	√						XL
crnand	√						XL
crnor	√						XL
cror	√						XL
crorc	√						XL
crxor	√						XL

Table A-45. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	64-Bit	Form
dcbf		√					X
dcbi[2]			√	√			X
dcbst		√					X
dcbt		√					X
dcbtst		√					X
dcbz		√					X
divd*x*[1]	√				√		XO
divdu*x*[1]	√				√		XO
divw*x*	√						XO
divwu*x*	√						XO
eciwx[3]		√				√	X
ecowx[3]		√				√	X
eieio		√					X
eqv*x*	√						X
extsb*x*	√						X
extsh*x*	√						X
extsw*x*[1]	√				√		X
fabs*x*	√						X
fadd*x*	√						A
fadds*x*	√						A
fcfid*x*[1]	√				√		X
fcmpo	√						X
fcmpu	√						X
fctid*x*[1]	√				√		X
fctidz*x*[1]	√				√		X
fctiw*x*	√						X
fctiwz*x*	√						X
fdiv*x*	√						A
fdivs*x*	√						A
fmadd*x*	√						A
fmadds*x*	√						A
fmr*x*	√						X
fmsub*x*	√						A
fmsubs*x*	√						A
fmul*x*	√						A

Table A-45. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	64-Bit	Form
fmuls*x*	√						A
fnabs*x*	√						X
fneg*x*	√						X
fnmadd*x*	√						A
fnmadds*x*	√						A
fnmsub*x*	√						A
fnmsubs*x*	√						A
fres*x* [3]	√					√	A
frsp*x*	√						X
frsqrte*x* [3]	√					√	A
fsel*x* [3]	√					√	A
fsqrt*x* [3]	√					√	A
fsqrts*x* [3]	√					√	A
fsub*x*	√						A
fsubs*x*	√						A
icbi		√					X
icbt [6]			√				X
isync		√					XL
lbz	√						D
lbzu	√						D
lbzux	√						X
lbzx	√						X
ld [1]	√				√		DS
ldarx [1]	√				√		X
ldu [1]	√				√		DS
ldux [1]	√				√		X
ldx [1]	√				√		X
lfd	√						D
lfdu	√						D
lfdux	√						X
lfdx	√						X
lfs	√						D
lfsu	√						D
lfsux	√						X
lfsx	√						X

Table A-45. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	64-Bit	Form
lha	√						D
lhau	√						D
lhaux	√						X
lhax	√						X
lhbrx	√						X
lhz	√						D
lhzu	√						D
lhzux	√						X
lhzx	√						X
lmw[4]	√						D
lswi [4]	√						X
lswx [4]	√						X
lwa [1]	√				√		DS
lwarx	√						X
lwaux [1]	√				√		X
lwax [1]	√				√		X
lwbrx	√						X
lwz	√						D
lwzu	√						D
lwzux	√						X
lwzx	√						X
mcrf	√						XL
mcrfs	√						X
mcrxr	√						X
mfcr	√						X
mffs*x*	√						X
mfmsr [2]			√	√			X
mfspr [5]	√		√	√			XFX
mfsr [2]			√	√			X
mfsrin [2]			√	√			X
mftb		√					XFX
mtcrf	√						XFX
mtfsb0*x*	√						X
mtfsb1*x*	√						X
mtfsf*x*	√						XFL

Table A-45. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	64-Bit	Form
mtfsfi*x*	√						X
mtmsr [2]			√	√			X
mtspr [5]	√		√	√			XFX
mtsr [2]			√	√			X
mtsrin [2]			√	√			X
mulhd*x* [1]	√				√		XO
mulhdu*x* [1]	√				√		XO
mulhw*x*	√						XO
mulhwu*x*	√						XO
mulld*x* [1]	√				√		XO
mulli	√						D
mullw*x*	√						XO
nand*x*	√						X
neg*x*	√						XO
nor*x*	√						X
or*x*	√						X
orc*x*	√						X
ori	√						D
oris	√						D
rfi [2]			√	√			XL
rldcl*x* [1]	√				√		MDS
rldcr*x* [1]	√				√		MDS
rldic*x* [1]	√				√		MD
rldicl*x* [1]	√				√		MD
rldicr*x* [1]	√				√		MD
rldimi*x* [1]	√				√		MD
rlwimi*x*	√						M
rlwinm*x*	√						M
rlwnm*x*	√						M
sc	√		√				SC
slbia [1, 2, 3]			√	√	√	√	X
slbie [1, 2, 3]			√	√	√	√	X
sld*x* [1]	√				√		X
slw*x*	√						X
srad*x* [1]	√				√		X

Table A-45. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	64-Bit	Form
sradi*x* [1]	√				√		XS
sraw*x*	√						X
srawi*x*	√						X
srd*x* [1]	√				√		X
srw*x*	√						X
stb	√						D
stbu	√						D
stbux	√						X
stbx	√						X
std [1]	√				√		DS
stdcx. [1]	√				√		X
stdu [1]	√				√		DS
stdux [1]	√				√		X
stdx [1]	√				√		X
stfd	√						D
stfdu	√						D
stfdux	√						X
stfdx	√						X
stfiwx [3]	√					√	X
stfs	√						D
stfsu	√						D
stfsux	√						X
stfsx	√						X
sth	√						D
sthbrx	√						X
sthu	√						D
sthux	√						X
sthx	√						X
stmw [4]	√						D
stswi [4]	√						X
stswx [4]	√						X
stw	√						D
stwbrx	√						X
stwcx.	√						X
stwu	√						D

Table A-45. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	64-Bit	Form
stwux	√						X
stwx	√						X
subf*x*	√						XO
subfc*x*	√						XO
subfe*x*	√						XO
subfic	√						D
subfme*x*	√						XO
subfze*x*	√						XO
sync	√						X
td [1]	√				√		X
tdi [1]	√				√		D
tlbia [2, 3]			√	√		√	X
tlbie [2, 3]			√	√		√	X
tlbld [2, 6]				√			X
tlbli [2, 6]				√			X
tlbsync [2, 3]			√	√			X
tw	√						X
twi	√						D
xor*x*	√						X
xori	√						D
xoris	√						D

1 64-bit instruction
2 Supervisor-level instruction
3 Optional in the PowerPC architecture
4 Load and store string or multiple instruction
5 Supervisor- and user-level instruction
6 e300 core-implementation specific instruction

Appendix B
Instructions Not Implemented

This appendix provides a list of the 32- and 64-bit instructions that are not implemented in the e300 core. It also provides a list of the 64-bit SPR encodings not implemented by the e300. Note that any attempt to execute unimplemented instructions generates an illegal instruction exception.

Table B-1 lists the 32-bit instructions that are optional to the PowerPC architecture and not implemented by the e300 core.

Table B-1. 32-Bit Instructions Not Implemented by the e300 core

Mnemonic	Instruction
eciwx	External Control In Word Indexed
ecowx	External Control Out Word Indexed
fsqrt	Floating Square Root (Double-Precision)
fsqrts	Floating Square Root Single
tlbia	TLB Invalidate All

Table B-2 provides a list of 64-bit instructions that are not implemented by 32-bit implementation such as the e300 core.

Table B-2. 64-Bit Instructions Not Implemented by the e300 core

Mnemonic	Instruction
cntlzd	Count Leading Zeros Double Word
divd	Divide Double Word
divdu	Divide Double Word Unsigned
extsw	Extend Sign Word
fcfid	Floating Convert From Integer Double Word
fctid	Floating Convert to Integer Double Word
fctidz	Floating Convert to Integer Double Word with Round toward Zero
ld	Load Double Word
ldarx	Load Double Word and Reserve Indexed
ldu	Load Double Word with Update
ldux	Load Double Word with Update Indexed
ldx	Load Double Word Indexed
lwa	Load Word Algebraic

Table B-2. 64-Bit Instructions Not Implemented by the e300 core (continued)

Mnemonic	Instruction
lwaux	Load Word Algebraic with Update Indexed
lwax	Load Word Algebraic Indexed
mulld	Multiply Low Double Word
mulhd	Multiply High Double Word
mulhdu	Multiply High Double Word Unsigned
rldcl	Rotate Left Double Word then Clear Left
rldcr	Rotate Left Double Word then Clear Right
rldic	Rotate Left Double Word Immediate then Clear
rldicl	Rotate Left Double Word Immediate then Clear Left
rldicr	Rotate Left Double Word Immediate then Clear Right
rldimi	Rotate Left Double Word Immediate then Mask Insert
slbia	SLB Invalidate All
slbie	SLB Invalidate Entry
sld	Shift Left Double Word
srad	Shift Right Algebraic Double Word
sradi	Shift Right Algebraic Double Word Immediate
srd	Shift Right Double Word
std	Store Double Word
stdcx.	Store Double Word Conditional Indexed
stdu	Store Double Word with Update
stdux	Store Double Word Indexed with Update
stdx	Store Double Word Indexed
td	Trap Double Word
tdi	Trap Double Word Immediate

Table B-3 provides the 64-bit SPR encoding that is not implemented by the e300 core.

Table B-3. 64-Bit SPR Encoding Not Implemented by the e300 core

SPR			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
280	01000	11000	ASR	Supervisor

Appendix C
Revision History

This appendix provides a list of the major differences between the *e300 PowerPC™ Core Reference Manual*, Revision 0, and the *e300 Power Architecture™ Core Family Reference Manual*, Revision 3.

C.1 Changes from Revision 2 to Revision 3

Major changes from the *e300 PowerPC™ Core Reference Manual*, Revision 2, to the *e300 Power Architecture™ Core Family Reference Manual*, Revision 3, are as follows:

Section, Page	Changes
Throughout	Provide information pertaining to the e300c3 configuration.

C.2 Changes from Revision 1 to Revision 2

Major changes from the *e300 PowerPC™ Core Reference Manual*, Revision 1, to the *e300 Power Architecture™ Core Family Reference Manual*, Revision 2, are as follows:

Section, Page	Changes
Book	Add caveats that the e300c2 does not support floating-point operations.
	Remove references to a moded 32- or 64-bit data bus because that is a feature that will be phased out of e300 cores. The default now is a 64-bit data bus.
Chapter 1	Add Section 1.5, “Differences Between e300 Cores.”
	Add notes that the e300c2 improves integer instruction throughput and significantly improves multiply instructions.
1.0/1-1	Change wording to explain references to e300, e300c1 and e300c2.
1.1/1-1	Add sentence stating that the e300c2 significantly improves multiply instructions.
1.1/1-1	Add sentence stating that the e300c2 eliminates the FPU.
1.1/1-3	Add Figure 1-2, “e300c2 Core Block Diagram.”
1.3.3.2/1-22	Add paragraph for e300c2 implementation and replace Figure 1-3, “Data Cache Organization,” with two figures: “e300c1 Data Cache Organization,” and “e300c2 Data Cache Organization.”
1.3.4.2/1-26	In Table 1-2, add the following sentence to the description of exception conditions for “Floating-point unavailable”: “In the e300c2 core, any attempt to execute a floating-point instruction results in a floating-point unavailable exception.”
1.4.7/1-55	Remove paragraph describing bus arbitration scheme.
2.1/2-4	Add PVR value of the e300c2 core in the bullet describing the PVR.

2.1/2-5	In Figure 2-2, “e300c1 Processor Version Register,” add PVR value of the e300c2 core in the PVR register diagram.
2.1/2-5	In Figure 2-3, “Machine State Register,” change reset value from “All zeros” to “0000_0040 or 0000_0000 or 0001_0041 or 0001_0001,” to reflect the values during different reset states.
2.1/2-6/2-7	In Table 2-3, “MSR Bit Settings,” add statements that bits FP, FE0, and FE1 are read-only in e300c2.
2.2.1/2-11	In Table 2-4, “e300 HID0 Field Descriptions,” add description to bit 25 to show the decrementer auto reload (DECAREN) bit found in e300c2 only.
	In Table 2-4, “e300 HID0 Field Descriptions,” add phrase in HID0[ICE] and HID0[ILOCK] descriptions to note that burst transactions can be generated even if the instruction cache is off or locked.
	In Table 2-4, “e300 HID0 Field Descriptions,” before the bit settings, add the following: “The $\overline{qreq}$ signal is asserted to indicate that the processor is ready to enter nap mode. If the system logic determines that the processor may enter nap mode, the quiesce acknowledge signal, $\overline{qack}$, is asserted to notify the processor.”
2.2.3/2-15	Add description to bit 11 in Table 2-7, “e300 HID2 Field Descriptions,” to show the enable weighted LRU (ELRW) bit found in e300c2 only.
	Add description to bit 12 in Table 2-7, “e300 HID2 Field Descriptions,” to show the no kill for snoop (NOKS) bit found in e300c2.
	Add differences between e300c1 and e300c2 in the instruction cache way-locking feature to HID2[16-18] in Table 2-7, “e300 HID2 Field Descriptions.”
	Add differences between e300c1 and e300c2 in the data cache way-locking feature to HID2[24-26] in Table 2-7, “e300 HID2 Field Descriptions.”
3.2.4.3.1/3-17	In the last sentence of the section, add the caveat that when HID0[IFEM] is set, the core broadcasts the M bit.
4.1.1/4-1	Change second and third bullets to include e300c2 implementation (16-Kbyte, four-way).
4.2/4-3	Add paragraph for e300c2 implementation and replace Figure 4-1, “Data Cache Organization,” with two figures: “e300c1 Data Cache Organization,” and “e300c2 Data Cache Organization.”
4.3/4-4	Add paragraph for e300c2 implementation and replace Figure 4-2, “Instruction Cache Organization,” with two figures: “e300c1 Instruction Cache Organization,” and “e300c2 Instruction Cache Organization.”
4.5.2.8/4/20	Remove statement that **icbi** broadcasts to the bus, because **icbi** does not broadcast to the bus in the e300 core. Modify the parenthetical description of **icbi** to say, “invalidate the old instruction cache entry in this processor.”
4.6.7/4-23	Add paragraph and Table 4-5, “e300c2 PLRU Replacement Way Selection,” to show e300c2 PLRU implementation.

4.6.7/4-24	Revise Figure 4-5, “PLRU Replacement Algorithm,” with note that B0 = 0 leg is always taken on e300c2.
4.10.1/4-33	Add row for e300c2 cache organization in Table 4-9, “Cache Organization.”
4.10.3.1.4/4-36	Add comment in example code that the number of blocks for e300c2 is 0x200. Added comment that the example uses e300c1 value of 0x400.
4.10.3.1.7/4-38	Separate Table 4-15, “e300 Core DWLCK[0–2] Encodings,” into two tables: “e300c1 Core DWLCK[0–2] Encodings” and “e300c2 Core DWLCK[0–2] Encodings.”
4.10.3.2.6/4-43	Separate Table 4-18, “e300 Core IWLCK[0–2] Encodings,” into two tables: “e300c1 Core IWLCK[0–2] Encodings” and “e300c2 Core IWLCK[0–2] Encodings.”
5.1/5-4	In Table 5-2, “Interrupts and Exception Conditions,” change the decrementer interrupt description to say that it is triggered when DEC[0] changes from 0 to 1, not when DEC[31] changes from 0 to 1.
5.2.1.4/5-12	In Figure 5-6, “Machine State Register (MSR),” change reset value from “All zeros” to “0000_0040 or 0000_0000 or 0001_0041 or 0001_0001,” to reflect the values during different reset states.
5.2.1.4/5-13	In Table 5-8, “MSR Bit Settings,” add statements that bits FP, FE0, and FE1 are read-only in e300c2.
7.3.2.1/7-9	Because the e300 core supports the hit-under-cancel capability, replace the last sentence of the section with the following two sentences: “The instruction fetch cancel extension allows a new instruction fetch to be issued to the cache or to the bus if a cancelled instruction fetch is pending or active on the bus. This is also called hit-under-cancel capability.”
7.4.2/7-21	Before the last sentence in the first paragraph of the section, add the following sentence: “In the e300c2, however, each of the two execution units can execute one multiply for a total of two multiply instructions executed in parallel.” Add two figures, “Instruction Timing—Integer Execution in the e300c1Core,” and “Instruction Timing—Integer Execution in the e300c2” to show difference in integer instruction throughput between e300c1 and e300c2.
7.7/7-25	Modify Table 7-4, “Integer Instructions,” to include latency information for the e300c2 core.
8.2/8-7	Replace statement that the data bus can be selected to be 32 or 64 bits wide with the following: “The address bus is 32 bits wide and the data bus is 64 bits wide.”
8.1.1/8/2	In Figure 8-1, “Core Interface Signals,” add tlbisync signal to the diagram.
8.1.2/8-2	In Table 8-1, “Summary of Selected Internal Signals,” add tlbisync signal description to the I/O description table.

C.3 Changes From Revision 0 to Revision 1

Major changes to the *e300 PowerPC™ Core Reference Manual*, from Revision 0 to Revision 1, are as follows:

Section, Page	Changes
Book	Add preface.
	Remove references to softstop.
	Add indexing.
	Add “Appendix C: Revision History.”
	Change references of e300v1 to e300c1 to denote ‘e300 configuration 1’.
	Change references of e300 to e300c1 where an implementation-specific feature was explained.
	Change term ‘exception’ to ‘interrupt’, where applicable.
	Remove caveats that **dcbi** should not be used on e300.
	Change references to JTAG/COP to JTAG/debug.
	Add references to Instruction Cache Block Touch (**icbt**) instruction.
1.0/1-1	Change wording to describe references to e300, and e300c1.
Chapter 1	Change LRU to PLRU when describing cache replacement policy.
Chapter 4	Change LRU to PLRU when describing cache replacement policy.
1.3.1.7.2/1-17	Remove bullet describing the EAR register and the **eciwx**, **ecowx** instructions (no longer supported).
1.3.1.7.2/1-17	Include other address breakpoint registers as SPRs in the e300.
1.3.4.2, 1-24/1-25	Change Table 1-2 to show that ISI exceptions are not caused when an instruction fetch cannot be performed when the fetch accesses a direct-store segment, because direct-stores are no longer supported.
	Change Table 1-2 to show that **lmw** and **stmw** do not cause an alignment exception when in true little-endian mode.
	Change Table 1-2 to show that alignment exceptions are not caused when crossing into a direct-store segment because direct-stores are no longer supported.
1.3.7.2/1-30	Change signals description to reflect a simpler introduction to signals.
2.1.2.12/2-20	Change Table 2-10 description of manufacturer ID to Freescale from Motorola.
Chapter 2	Add statements that reserved bits should be cleared for future compatibility.
2.1.1/2-5	Before the description of the machine state register, add table entitled, “Assigned PVR Values,” to show PVR values of different processors.
2.1.1/2-6	Add footnote to Table 2-2, “MSR Bit Settings,” to show that reserved bits must be set to zero.
2.1.1/2-8	Remove bullet describing the EAR register and the **eciwx**, **ecowx** instructions (no longer supported).

2.1.2.3/2-14	Change the name of bit 7 in Table 2-6 from MESI to MESISTATE.
2.1.2.12/2-20	Change SVR value in Figure 2-17, “System Version Register (SVR),” to show that it is determined by the SoC.
3.2.4.3.6/3-20	Remove third bullet on page noting that **lmw** and **stmw** cause an alignment exception when in true little-endian mode.
3.2.5.3/3-28	Remove statement that cache control instructions that deal with direct-store segments are treated as no-ops since direct-stores are no longer supported.
3.2.5.3/3-28	Remove statements noting that incoherency may occur if a write-through store is followed by a **dcbz** instruction.
3.2.5.3/3-28	Remove statements noting that broadcasting a sequence of **dcbz** instructions may cause snoop accesses to be retired indefinitely.
3.2.6.3.1/3-32	Remove statement that **dcbi** causes a direct storage interrupt and added that **dcbi** is used to invalidate cache blocks.
Chapter 4	Revise some section titles for clarity.
4.2/4-3	Change first paragraph to specify e300c1 implementation.
4.3/4-4	Change first paragraph to specify e300c1 implementation.
4.6.7/4-23	Change second paragraph and Table 4-7 title to specify e300c1 PLRU implementation.
4.10.3.1.7/4-38	In Table 4-17, relabel “Ways Locked” column as “e300c1 Ways Locked.”
4.10.3.2.6/4-41	In Table 4-20, relabel “Ways Locked” column as “e300c1 Ways Locked.”
Chapter 5	Remove references to direct-stores (no longer supported).
5.1/5-4	Remove **lmw** and **stmw** from list of instructions mentioned in Table 5-2 that cause an alignment exception when in little-endian mode.
5.1.1/5-6	Remove mention of power-on-reset as a cause for system reset in Table 5-3. $\overline{\textit{hreset}}$ is very similar to power-on-reset and is the only listed cause of system reset now.
5.1.1/5-7	Remove **lmw** and **stmw** from list of instructions mentioned in Table 5-3 that cause an alignment exception when in little-endian mode.
5.2.1.1/5-10	Following Table 5-4, add table entitled, “Bit Settings for Program Interrupts,” to show SRR1 bit settings on a program interrupt.
5.2.1.2/5-11	Change statement that mentioned CSRR1 loading only bits 16–31 from MSR to loading all bits [0–31] from MSR for the e300 core.
5.5.2/5-21	Add statement that the $\overline{\textit{tea}}$ signal is still monitored even when HID0[EMCP] is cleared.
5.5.3/5-24	Change statement describing a DSI interrupt caused by a translation error or protection violation. Instead of stating that the DAR points to the byte address of the offending page, now states that entire instruction should be re-executed. This is due to a previously documented errata.
5.5.3/5-25	Remove load multiple instructions as instructions that can be partially executed on a DSI interrupt.

5.5.4/5-25	Remove statement that an ISI interrupt occurs when an attempt is made to fetch an instruction from guarded memory when MSR[IR]=1 because no ISI occurs with guarded bit on.
5.5.6/5-26	Remove **lmw** and **stmw** from list of instructions mentioned that cause an alignment exception when in little-endian mode.
5.0/5-1	Add reference to CSRR0, CSRR1 as save/restore registers for critical interrupts.
5.1/5-4	Remove references to direct-store segments from Table 5-2.
5.2.1.1/5-10	Change Table 5-4 on SRR1 bit settings on an MCE to match the settings on Table 5-14.
5.5.3/5-23	Remove mention of direct stores in description of bit 0 of DSISR in Table 5-14.
Chapter 6	Change terminology to Reference and Change bits (from Referenced and Changed bits)
10.1/10-1	Add mention of address breakpoint interrupt and DSI interrupt when breakpoint conditions are met.
10.1.6/10-3	Remove Software Debug section as it was beyond the scope of this document.
10.4/10-5	Change IABR_ADDR variable to IABR[CEA], and IABR2_ADDR variable to IABR2[CEA], to eliminate redundant information.
10.2.1/10-4	Remove Breakpoint Enabled section as it was redundant with other information in this chapter.

Glossary

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

A

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible *implementations*.

Asynchronous interrupt. *interrupts* that are caused by events external to the processor's execution. In this document, the term *asynchronous interrupt* is used interchangeably with the word *interrupt*.

Atomic access. A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to the fact that the transactions are indivisible). The PowerPC architecture implements *atomic accesses* through the **lwarx**/**stwcx.** instruction pair.

B

BAT (block address translation) mechanism. A software-controlled array that stores the available block address translations on-chip.

Beat. A single state on the e300 bus interface that may extend across multiple bus cycles. A e300 transaction can be composed of multiple address or data *beats*.

Biased exponent. An *exponent* whose range of values is shifted by a constant (bias). Typically a bias is provided to allow a range of positive values to express a range that includes both positive and negative values.

Big-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *most-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the *most-significant byte*. See *Little-endian*.

Block. An area of memory that ranges from 128 Kbytes to 256 Mbytes whose size, translation, and protection attributes are controlled by the *BAT* mechanism.

Boundedly undefined. A characteristic of certain operation results that are not rigidly prescribed by the PowerPC architecture. Boundedly- undefined results for a given operation may vary among implementations and between execution attempts in the same implementation.

Although the architecture does not prescribe the exact behavior for when results are allowed to be *boundedly undefined*, the results of executing instructions in contexts where results are allowed to be *boundedly undefined* are constrained to ones that could have been achieved by executing an arbitrary sequence of defined instructions, in valid form, starting in the state the machine was in before attempting to execute the given instruction.

Branch folding. The replacement with target instructions of a branch instruction and any instructions along the not-taken path when a branch is either taken or predicted as taken.

Branch prediction. The process of guessing whether a branch will be taken. Such predictions can be correct or incorrect; the term 'predicted' as it is used here does not imply that the prediction is correct (successful). The PowerPC architecture defines a means for *static branch* prediction as part of the instruction encoding.

Branch resolution. The determination of whether a branch is taken or not taken. A branch is said to be resolved when the processor can determine which instruction path to take. If the branch is resolved as predicted, the instructions following the predicted branch that may have been speculatively executed can complete (see *Completion*). If the branch is not resolved as predicted, instructions on the mispredicted path, and any results of speculative execution, are purged from the pipeline and fetching continues from the nonpredicted path.

Burst. A multiple-beat data transfer whose total size is typically equal to a cache block.

Bus clock. Clock that causes the bus state transitions.

Bus master. The owner of the address or data bus; the device that initiates or requests the transaction.

C

Cache. High-speed memory containing recently accessed data or instructions (subset of main memory).

Cache block. A small region of contiguous memory that is copied from memory into a *cache*. The size of a *cache block* may vary among processors; the maximum block size is one *page*. In PowerPC processors, *cache coherency* is maintained on a cache-block basis. Note that the term *cache block* is often used interchangeably with 'cache line.'

Cache coherency. An attribute wherein an accurate and common view of memory is provided to all devices that share the same memory system. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor's cache.

Cache flush. An operation that removes from a cache any data from a specified address range. This operation ensures that any modified data within the specified address range is written back to main memory. This operation is generated typically by a Data Cache Block Flush (**dcbf**) instruction.

Caching-inhibited. A memory update policy in which the cache is bypassed and the load or store is performed to or from main memory.

Cast out. A *cache block* that must be written to memory when a cache miss causes a *cache block* to be replaced.

Changed bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the changed bit if any store is performed into the *page*. See also *Page access history bits* and *Referenced bit*.

Clean. An operation that causes a *cache block* to be written to memory, if modified, and then left in a valid, unmodified state in the cache.

Clear. To cause a bit or bit field to register a value of zero. See also *Set*.

Completion. Completion occurs when an instruction has finished executing, written back any results, and is removed from the completion queue (CQ). When an instruction completes, it is guaranteed that this instruction and all previous instructions can cause no interrupts.

Context synchronization. An operation that ensures that all instructions in execution complete past the point where they can produce an *interrupt*, that all instructions in execution complete in the context in which they began execution, and that all subsequent instructions are *fetched* and executed in the new context. Context synchronization may result from executing specific instructions (such as **isync** or **rfi**) or when certain events occur (such as an *interrupt*).

Copy-back operation. A cache operation in which a cache line is copied back to memory to enforce cache coherency. Copy-back operations consist of snoop push-out operations and cache cast-out operations.

D

Denormalized number. A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

Direct-mapped cache. A cache in which each main memory address can appear in only one location within the cache, operates more quickly when the memory request is a cache hit.

E

Effective address (EA). The 32-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* address or an I/O address.

Exception. A condition that, if enabled, generates an interrupt.

Exclusive state. MEI state (E) in which only one caching device contains data that is also in system memory.

Execution synchronization. A mechanism by which all instructions in execution are architecturally complete before beginning execution (appearing to begin execution) of the next instruction. Similar to context synchronization but doesn't force the contents of the instruction buffers to be deleted and refetched.

Exponent. In the binary representation of a floating-point number, the exponent is the component that normally signifies the integer power to which the value two is raised in determining the value of the represented number. See also *Biased exponent*.

F

Fall-through (branch fall-through). A not-taken branch. On the e300 core, fall-through branch instructions are removed from the instruction stream at dispatch. That is, these instructions are allowed to fall through the instruction queue through the dispatch mechanism, without either being passed to an execution unit and or given a position in the CQ.

Feed-forwarding. A e300 feature that reduces the number of clock cycles that an execution unit must wait to use a register. When the source register of the current instruction is the same as the destination register of the previous instruction, the result of the previous instruction is routed to the current instruction at the same time that it is written to the register file. With feed-forwarding, the destination bus is gated to the waiting execution unit over the appropriate source bus, saving the cycles which would be used for the write and read.

Fetch. Retrieving instructions from either the cache or main memory and placing them into the instruction queue.

Finish. Finishing occurs in the last cycle of execution. In this cycle, the CQ entry is updated to indicate that the instruction has finished executing.

Floating-point register (FPR). Any of the 32 registers in the floating-point register file. These registers provide the source operands and destination results for floating-point instructions. Load instructions move data from memory to FPRs and store instructions move data from FPRs to memory. The FPRs are 64 bits wide and store floating-point values in double-precision format.

Floating-point unit. The functional unit in the e300c1e300 processor responsible for executing all floating-point instructions.

Flush. An operation that causes a cache block to be invalidated and the data, if modified, to be written to memory.

Folding. See *Branch folding*.

Fraction. In the binary representation of a floating-point number, the field of the *significand* that lies to the right of its implied binary point.

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

Guarded. The guarded attribute pertains to out-of-order execution. When a page is designated as guarded, instructions and data cannot be accessed out-of-order.

H

Harvard architecture. An architectural model featuring separate caches and other memory management resources for instructions and data.

Hashing. An algorithm used in the *page table* search process.

I

IEEE 754. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point numbers.

Illegal instructions. A class of instructions that are not implemented for a particular PowerPC processor. These include instructions not defined by the PowerPC architecture. In addition, for 32-bit implementations, instructions that are defined only for 64-bit implementations are considered to be illegal instructions. For 64-bit implementations instructions that are defined only for 32-bit implementations are considered to be illegal instructions.

Implementation. A particular processor that conforms to the PowerPC architecture, but may differ from other architecture-compliant implementations for example in design, feature set, and implementation of *optional* features. The PowerPC architecture has many different implementations.

Imprecise interrupt. A type of *synchronous interrupt* that is allowed not to adhere to the precise interrupt model (see *Precise interrupt*). The PowerPC architecture allows only floating-point exceptions to be handled imprecisely.

Instruction queue. A holding place for instructions fetched from the current instruction stream.

Integer unit. The functional unit in the e300 responsible for executing all integer instructions.

In-order. An aspect of an operation that adheres to a sequential model. An operation is said to be performed in-order if, at the time that it is performed, it is known to be required by the sequential execution model. See *Out-of-order.*

Instruction latency. The total number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

Interrupt. A condition encountered by the processor that requires special, supervisor-level processing.

Interrupt handler. A software routine that executes when an interrupt is taken. Normally, the interrupt handler corrects the condition that caused the interrupt, or performs some other meaningful task (that may include aborting the program that caused the interrupt). The address for each interrupt handler is identified by an interrupt vector offset defined by the architecture and a prefix selected via the MSR.

K

Key bits. A set of key bits referred to as Ks and Kp in each segment register and each BAT register. The key bits determine whether supervisor or user programs can access a *page* within that *segment* or *block*.

Kill. An operation that causes a *cache block* to be invalidated without writing any modified data to memory.

L

Latency. The number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction.

L2 cache. See *Secondary cache*.

Least-significant bit (lsb). The bit of least value in an address, register, field, data element, or instruction encoding.

Least-significant byte (LSB). The byte of least value in an address, register, data element, or instruction encoding.

Little-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *least-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the *most-significant byte*. See *Big-endian*.

M

Mantissa. The decimal part of logarithm.

MEI (modified/exclusive/invalid). *Cache coherency* protocol used to manage caches on different devices that share a memory system. Note that the PowerPC architecture does not specify the implementation of a MEI protocol to ensure cache coherency.

Memory access ordering. The specific order in which the processor performs load and store memory accesses and the order in which those accesses complete.

Memory-mapped accesses. Accesses whose addresses use the page or block address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.

Memory coherency. An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.

Memory consistency. Refers to agreement of levels of memory with respect to a single processor and system memory (for example, on-chip cache, secondary cache, and system memory).

Memory management unit (MMU). The functional unit that is capable of translating an *effective* (logical) *address* to a physical address, providing protection mechanisms, and defining caching methods.

Modified state. MEI state (M) in which one, and only one, caching device has the valid data for that address. The data at this address in external memory is not valid.

Most-significant bit (msb). The highest-order bit in an address, registers, data element, or instruction encoding.

Most-significant byte (MSB). The highest-order byte in an address, registers, data element, or instruction encoding.

N

NaN. An abbreviation for not a number; a symbolic entity encoded in floating-point format. There are two types of NaNs—signaling NaNs and quiet NaNs.

No-op. No-operation. A single-cycle operation that does not affect registers or generate bus activity.

Normalization. A process by which a floating-point value is manipulated such that it can be represented in the format for the appropriate precision (single- or double-precision). For a floating-point value to be representable in the single- or double-precision format, the leading implied bit must be a 1.

O

OEA (operating environment architecture). The level of the architecture that describes PowerPC memory management model, supervisor-level registers, synchronization requirements, and the interrupt model. It also defines the time-base feature from a supervisor-level perspective. Implementations that conform to the PowerPC OEA also conform to the PowerPC UISA and VEA.

Optional. A feature, such as an instruction, a register, or an interrupt, that is defined by the PowerPC architecture but not required to be implemented.

Out-of-order. An aspect of an operation that allows it to be performed ahead of one that may have preceded it in the sequential model, for example, speculative operations. An operation is said to be performed out-of-order if, at the time that it is performed, it is not known to be required by the sequential execution model. See *In-order.*

Out-of-order execution. A technique that allows instructions to be issued and completed in an order that differs from their sequence in the instruction stream.

Overflow. An condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are multiplied, the result may not be representable in 32 bits. Since the 32-bit registers of the e300 cannot represent this sum, an overflow condition occurs.

P

Page. A region in memory. The OEA defines a page as a 4-Kbyte area of memory, aligned on a 4-Kbyte boundary.

Page access history bits. The *changed* and *referenced* bits in the PTE keep track of the access history within the page. The referenced bit is set by the MMU whenever the page is accessed for a read or write operation. The changed bit is set when the page is stored into. See *Changed bit* and *Referenced bit*.

Page fault. A page fault is a condition that occurs when the processor attempts to access a memory location that does not reside within a *page* not currently resident in *physical memory.* On PowerPC processors, a page fault interrupt condition occurs when a matching, valid *page table entry* (PTE[V] = 1) cannot be located.

Page table. A table in memory is comprised of *page table entries*, or PTEs. It is further organized into eight PTEs per PTEG (page table entry group). The number of PTEGs in the page table depends on the size of the page table (as specified in the SDR1 register).

Page table entry (PTE). Data structures containing information used to translate *effective address* to physical address on a 4-Kbyte page basis. A PTE consists of 8 bytes of information in a 32-bit processor and 16 bytes of information in a 64-bit processor.

Park. The act of allowing a bus master to maintain bus mastership without having to arbitrate.

Physical memory. The actual memory that can be accessed through the system's memory bus.

Pipelining. A technique that breaks operations, such as instruction processing or bus transactions, into smaller distinct stages or tenures (respectively) so that a subsequent operation can begin before the previous one has completed.

Precise interrupts. A category of interrupt for which the pipeline can be stopped so instructions that preceded the faulting instruction can complete and subsequent instructions can be flushed and redispatched after interrupt handling has completed. See *Imprecise interrupts*.

Primary opcode. The most-significant 6 bits (bits 0–5) of the instruction encoding that identifies the type of instruction.

Program order. The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.

Protection boundary. A boundary between *protection domains*.

Protection domain. A protection domain is a segment, a virtual page, a BAT area, or a range of unmapped effective addresses. It is defined only when the appropriate relocate bit in the MSR (IR or DR) is 1.

Q

Quiesce. To come to rest. The processor is said to quiesce when an interrupt is taken or a **sync** instruction is executed. The instruction stream is stopped at the decode stage and executing instructions are allowed to complete to create a controlled context for instructions that may be affected by out-of-order, parallel execution. See *Context synchronization*.

Quiet NaN. A type of *NaN* that can propagate through most arithmetic operations without signaling interrupts. A quiet NaN is used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid. See *Signaling NaN*.

R

rA. The **r**A instruction field is used to specify a GPR to be used as a source or destination.

rB. The **r**B instruction field is used to specify a GPR to be used as a source.

rD. The **r**D instruction field is used to specify a GPR to be used as a destination.

rS. The **r**S instruction field is used to specify a GPR to be used as a source.

Real address mode. An MMU mode when no address translation is performed and the *effective address* specified is the same as the physical address. The processor's MMU is operating in real address mode if its ability to perform address translation has been disabled through the MSR registers IR and/or DR bits.

Record bit. Bit 31 (or the Rc bit) in the instruction encoding. When it is set, updates the condition register (CR) to reflect the result of the operation.

Referenced bit. One of two *page history bits* found in each *page table entry.* The processor sets the *referenced bit* whenever the page is accessed for a read or write. See also *Page access history bits*.

Register indirect addressing. A form of addressing that specifies one GPR that contains the address for the load or store.

Register indirect with immediate index addressing. A form of addressing that specifies an immediate value to be added to the contents of a specified GPR to form the target address for the load or store.

Register indirect with index addressing. A form of addressing that specifies that the contents of two GPRs be added together to yield the target address for the load or store.

Rename register. Temporary buffers used by instructions that have finished execution but have not completed.

Reservation. The processor establishes a reservation on a *cache block* of memory space when it executes an **lwarx** instruction to read a memory semaphore into a GPR.

Reservation station. A buffer between the dispatch and execute stages that allows instructions to be dispatched even though the results of instructions on which the dispatched instruction may depend are not available.

Retirement. Removal of the completed instruction from the CQ.

RISC (reduced instruction set computing). An *architecture* characterized by fixed-length instructions with nonoverlapping functionality and by a separate set of load and store instructions that perform memory accesses.

S

Scan interface. The e300 test interface.

Secondary cache. A cache memory that is typically larger and has a longer access time than the primary cache. A secondary cache may be shared by multiple devices. Also referred to as L2, or level-2, cache.

Set (*v*)**.** To write a nonzero value to a bit or bit field; the opposite of *clear*. The term 'set' may also be used to generally describe the updating of a bit or bit field.

Set (*n*)**.** A subdivision of a *cache*. Cacheable data can be stored in a given location in one of the sets, typically corresponding to its lower-order address bits. Because several memory locations can map to the same location, cached data is typically placed in the set whose *cache block* corresponding to that address was used least recently. See *Set-associative*.

Set-associative. Aspect of cache organization in which the cache space is divided into sections, called *sets*. The cache controller associates a particular main memory address with the contents of a particular set, or region, within the cache.

Shadowing. Shadowing allows a register to be updated by instructions that are executed out of order without destroying machine state information.

Signaling NaN. A type of *NaN* that generates an invalid operation program interrupt when it is specified as arithmetic operands. See *Quiet NaN*.

Significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

Simplified mnemonics. Assembler mnemonics that represent a more complex form of a common operation.

Slave. The device addressed by a master device. The slave is identified in the address tenure and is responsible for supplying or latching the requested data for the master during the data tenure.

Snooping. Monitoring addresses driven by a bus master to detect the need for coherency actions.

Snoop push. Response to a snooped transaction that hits a modified cache block. The cache block is written to memory and made available to the snooping device.

Split-transaction. A transaction with independent request and response tenures.

Split-transaction bus. A bus that allows address and data transactions from different processors to occur independently.

Stage. The term *stage* is used in two different senses, depending on whether the pipeline is being discussed as a physical entity or a sequence of events. In the latter case, a stage is an element in the pipeline during which certain actions are performed, such as decoding the instruction, performing an arithmetic operation, or writing back the results. Typically, the latency of a stage is one processor clock cycle. Some events, such as dispatch, write-back, and completion, happen instantaneously and may be thought to occur at the end of a stage. An instruction can spend multiple cycles in one stage. An integer multiply, for example, takes multiple cycles in the execute stage. When this occurs, subsequent instructions may stall. An instruction may also occupy more than one stage simultaneously, especially in the sense that a stage can be seen as a physical resource—for example, when instructions are dispatched they are assigned a place in the CQ at the same time they are passed to the execute stage. They can be said to occupy both the complete and execute stages in the same clock cycle.

Stall. An occurrence when an instruction cannot proceed to the next stage.

Static branch prediction. Mechanism by which software (for example, compilers) can hint to the machine hardware about the direction a branch is likely to take.

Store Queue. Holds store operations that have not been committed to memory, resulting from completed or retired instructions.

Superscalar. A superscalar processor is one that can dispatch multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can be in the same stage at the same time.

Supervisor mode. The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and can access the supervisor memory space, among other privileged operations.

Synchronization. A process to ensure that operations occur strictly *in order.* See *Context synchronization* and *Execution synchronization*.

Synchronous interrupt. An *interrupt* that is generated by the execution of a particular instruction or instruction sequence. There are two types of synchronous interrupts, *precise* and *imprecise*.

System memory. The physical memory available to a processor.

T

Tenure. The period of bus mastership. For the e300, there can be separate address bus tenures and data bus tenures. A tenure consists of three phases: arbitration, transfer, and termination.

TLB (translation lookaside buffer). A cache that holds recently-used *page table entries*.

Throughput. The measure of the number of instructions that are processed per clock cycle.

Transaction. A complete exchange between two bus devices. A transaction is typically comprised of an address tenure and one or more data tenures, which may overlap or occur separately from the address tenure. A transaction may be minimally comprised of an address tenure only.

Transfer termination. Signal that refers to both signals that acknowledge the transfer of individual beats (of both single-beat transfer and individual beats of a burst transfer) and to signals that mark the end of the tenure.

U

UISA (user instruction set architecture). The level of the architecture to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions and interrupt model as seen by user programs, and the memory and programming models.

Underflow. A condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result requires a smaller *exponent* and/or *mantissa* than the single-precision format can provide. In other words, the result is too small to be represented accurately.

User mode. The operating state of a processor used typically by application software. In user mode, software can access only certain control registers and can access only user memory space. No privileged operations can be performed. Also referred to as problem state.

V

VEA (virtual environment architecture). The level of the *architecture* that describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, defines cache control instructions, and defines the time-base facility from a user-level perspective. *Implementations* that conform to the PowerPC VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

Virtual address. An intermediate address used in the translation of an *effective address* to a physical address.

Virtual memory. The address space created using the memory management facilities of the processor. Program access to *virtual memory* is possible only when it coincides with *physical memory.*

W

Way. A location in the cache that holds a cache block, its tags and status bits.

Word. A 32-bit data element.

Write-back. A cache memory update policy in which processor write cycles are directly written only to the cache. External memory is updated only indirectly, for example, when a modified cache block is *cast out* to make room for newer data.

Write-through. A cache memory update policy in which all processor write cycles are written to both the cache and memory.

Index

A

Accesses, core interface, overview, 8-7
Address calculation
  branch instruction, 3-23
  effective address, 3-8
Address generation
  floating-point load/store, 3-21
  integer load/store, 3-17
  physical address generation, MMU, 6-1
    see also Memory management unit (MMU), 6-1
Address translation, *see* Memory management unit (MMU)
Addressing
  memory, 3-7
  memory operand, 3-1
  modes, 3-7
Alignment
  interrupt, 5-25, 6-15
    see also Interrupt handling, 5-4
  misaligned accesses, 3-2
  overview, 1-28
Architecture, PowerPC, xxiii, 1-15
Assembly language programs, simplified mnemonics, 3-33
Asynchronous interrupts
  maskable, 5-2, 5-5
  nonmaskable, 5-2, 5-5
Atomic memory references, using **lwarx/stwcx.**, 4-14
Automatic power reduction mode, 9-1

B

Base/decrementer registers, 9-2
Big-endian mode, 5-14
Block address translation (BAT), 1-5, 6-19
  BAT registers and cache locking implications, 4-35, 4-40
  block address translation flow, 6-11
  see also Memory Management Unit (MMU), 1-5
  selection of block address translation, 6-8
Block diagram, e300 core, 1-2
Boundedly undefined, definition, 3-5
Branch instructions
  address calculation, 3-23
  branch and flow control instructions, 3-22
  condition register logical, A-22
  summary, A-22
  system linkage, A-22
  tracing on branch instructions, 5-13, 5-32, 10-3, 10-4
  trap, A-23
Branch processing unit (BPU), 1-1, 7-3
  branch folding, 7-1, 7-18
  branch instruction timing, 7-18, 7-21
    latency, 7-28
  branch prediction, static, 7-1, 7-19
  branch resolution
    definition, 7-1
    resource requirements, 7-26
  overview, 1-9
Breakpoints
  address matching options, 10-4
  branch trace enabling, 5-13, 5-32, 10-1, 10-3, 10-4
  data address breakpoints
    DSI (data storage interrupt), 10-3, 10-4
    match conditions, 10-2
  instruction address breakpoints, IABR interrupt, 10-1
  signaling, 1-33
  single-stepping, 10-1, 10-3
    single-step trace enabling, 5-13, 5-32, 10-3, 10-4
  using breakpoints, 10-3
Bus interface unit (BIU), 1-12, 4-2
Bus snooping, 9-2
Byte ordering, 3-1
  byte ordering considerations, 5-20
  interrupt LE mode (MSR[ILE] bit), 5-12
  LE (or BE) mode (MSR[LE] bit), 5-14
Byte-reverse instructions, A-20

C

Cache-inhibited accesses (I bit), *see* Memory/cache access attributes (WIMG bits)
Caches
  bus interface buffers, 4-26–4-27
    address pipelining, 4-26
    load-ahead-of-store capability, 4-26
    pipeline collision detection, 4-26
    reservation address snooping for **lwarx/stwcx.**, 4-26
  cache control
    and CSB operations, 4-29, 8-7
    broadcasting operations, 4-17
    cache control instructions, 4-17–4-21
    enabling/disabling caches
      data cache, 4-14
      instruction cache, 4-16
    invalidating caches

data cache, 4-15
instruction cache, 4-16
parameters in HID*n*, 4-14–4-17
parity error reporting, 4-14
WIMG bits, *see* Memory/cache access attributes (WIMG bits), 4-14
cache locking
applications information, 4-32
data cache locking
guidelines, 4-35–4-38
overview, 4-15
entire cache locking, 4-32
instruction cache locking
guidelines, 4-40–4-44
overview, 4-16
way protection, 4-17
MSR bits and cache locking, 4-34
register summary, 4-33
terminology, 4-32
way locking, 1-25, 4-33
cache management instructions, 3-32, A-23
cache miss effects, 7-14
coherency, 4-5–4-14, 9-4
3-state (MEI) coherency model, 4-9–4-10
4-state (MESI) coherency model, 4-9–4-11
in single-processor systems, 4-12
load and store operations, 4-12
CSB (coherent system bus), 4-26, 4-27
instruction cache arbitration, 7-10
instruction cache hit timing, 7-11
loads and stores
caching-inhibited loads/stores, 4-13
operations, 1-12, 4-21–4-25
organization, 4-3, 4-4, 4-33
overview and features, 4-1–4-2
parity checking, 4-25
performance
cache-inhibited pages, 7-25
memory considerations, 7-24
performed loads and stores, definition, 4-13
PLRU replacement algorithm, 4-4
snooping, 4-29–4-32
way-locking, 1-25
Change (C) bit, 6-10, 6-20
checking, 6-36
maintenance recording, 6-10, 6-20–6-22
Checkstop signal, 8-9
Checkstop state, 5-22
Clock multiplier, 1-14
Clock signals, *pll_cfg*n, 8-4
Coherent system bus (CSB), 4-26, 4-27
overview, 8-1
parity checking, 8-9
signals described, 8-1–8-4
Compare instructions, A-16
Completion queue (CQ), 7-1, Glossary-3
Completion unit, overview, 1-10
Condition register (CR), 1-18
Context synchronization, 3-8
Conventions, xxvi, xxxii
Copy-back mode, 7-25
Core interface
accesses, overview, 8-7
checkstops, 8-9
core quiesce control signals, 8-9
external interrupts, 8-8
IEEE 1149.1-compliant interface, 8-9
reset inputs, 8-9
signal groupings, 8-1, 8-2
signal summary, 8-2
CR (condition register)
logical instructions, 3-23
overview, 1-18
Critical input (*cint*) interrupt, 1-28, 5-4
enabling with MSR[CE], 5-13, 5-30
CSRR*n* (critical interrupt save/restore regs. 0–1), 2-10, 2-11, 2-21–2-22, 5-8, 5-10, 5-15, 5-16, 5-17
CTR (count register), 2-6

D

DABR/DABR2 (data address breakpoint regs.), 2-11, 2-25, 10-2
DAR (data address register), 2-10, 5-23, 10-2, 10-3
Data access errors, *see* DSI (data storage interrupt)
Data cache, *see* Caches
Data errors, *see* Machine check interrupt
Data TLB miss on load interrupt, 1-29, 5-5, 5-33
Data TLB miss on store interrupt, 1-29, 5-5, 5-34
DBAT, *see* Block address translation (BAT)
DBAT*n*U/L (data block address translation regs. 0–7, upper/lower), 2-9, 2-11, 2-20, 2-21
DBCR (data address breakpoint control reg.), 2-11, 2-27, 10-2
dcbf, 4-19
dcbi, 4-19
dcbst, 4-19
dcbt, 4-18
dcbtst, 4-18
dcbz, 4-18
DCMP (data TLB compare register), 2-18, 6-32, 6-34
Debug facilities, 1-33
debugging software, 10-3
interrupt vectors for debugging, 10-3
other debug resources, 10-2

performance monitor uses, 1-14, 11-1
registers, 10-1–10-3
see also Breakpoints
single-stepping, 10-1, 10-3
tracing on branch instructions, 5-13, 5-32, 10-3, 10-4
using breakpoints, 10-3
DEC (decrementer register), 2-10
Decrementer, 1-28
exception, 9-2
interrupt, 5-4, 5-30, 9-1
timer, 9-1
Direct address translation (translation disabled)
data accesses, 6-11, 6-19
instruction accesses, 6-11, 6-19
see also Memory management unit (MMU), 6-19
DMISS (data TLB miss address reg.), 2-18, 6-31, 6-34
Doze mode, 9-2, 9-3
DSI (data storage interrupt), 1-27, 5-3, 5-23, 10-3, 10-4
DSISR (DSI status register), 2-10, 5-1, 10-2, 10-3
DTLB, 1-5
Dynamic power management, 9-1
Dynamic power management, modes, 9-2

E

e300 core, differences between cores, 1-34
e300-specific instructions, 3-34
e300-specific registers, 2-10–2-27
Effective address (EA)
calculation, 3-8
translation, *see* Memory management unit (MMU)
eieio, 4-14
Endian modes, 3-1
byte ordering
interrupt LE mode (MSR[ILE] bit), 5-12
Endian modes and byte ordering
little-endian mode (MSR[LE] bit), 5-14
Event counting, *see* Performance monitor APU
Exceptions
enabling and disabling interrupts and exceptions, 5-14
overview, 1-25
see also Interrupt handling
types (more granular than interrupts)
floating-point enabled exceptions (program interrupt), 5-4, 5-28
illegal instr. exception (program interrupt), 5-4, 5-28
privileged instr. exception (program interrupt), 5-4, 5-28
trap instr. exceptions (program interrupt), 5-4, 5-28
Execution synchronization, 3-9
Execution timing
cache-related latency
cache-inhibited pages, 7-25
i-cache arbitration, 7-10
i-cache hit, 7-11
definitions
branch folding, 7-1
branch prediction, 7-1
branch resolution, 7-1
completion, 7-1
finish, 7-1
latency, 7-1
retirement, 7-2
stall, 7-2
throughput, 7-2
examples
cache hit case, 7-12
cache miss case, 7-15, 7-22, 7-23, C-3
execution units, 7-3
branch processing unit (BPU), 7-18, 7-21
branch folding, 7-1, 7-18
branch prediction, 7-1, 7-19
branch resolution, 7-1, 7-26
floating-point unit (FPU), 7-23, 7-31
integer unit (IU), 7-3, 7-21, 7-29
load/store unit (LSU), 7-24, 7-32
system register unit (SRU), 7-24, 7-28, 7-29
instruction flow, 7-9
instruction latency summary, 7-28
instruction pipeline stages, 7-1, 7-2, 7-4, 7-5, 7-8
completion, 7-1, 7-2
completion considerations, 7-16
resource requirements, 7-27
dispatch
dispatch considerations, 7-16
resource requirements, 7-27
finish, 7-1
write-back, 7-2
instruction queue (IQ), 7-9
instruction scheduling guidelines, 7-26
memory performance considerations, 7-24
memory coherency required (M bit), 7-25
rename registers, 7-2, 7-16
reservation stations, 7-2
External input (*int*) interrupt, 5-4, 5-25
enabling with MSR[EE], 5-12
External system logic, 9-2

F

Finish cycle
definition, 7-1
see also Execution timing
Floating-point model
enabling (FP available, MSR[FP]), 5-13
exceptions
floating-point unavailable interrupt, 5-4, 5-29

FP interrupt mode 0, 5-13, 5-14
FP interrupt mode 1, 5-13
IEEE exceptions, program interrupt, 5-29
execution model, 3-3
floating-point execution unit (FPU), 7-3
execution timing, 7-23
latency, FP instructions, 7-31
FP arithmetic instructions, A-17
FP compare instructions, A-18
FP load instructions, A-21
FP move instructions, A-22
FP multiply-add instructions, A-18
FP registers (FPR*n*), 1-18
FP rounding/conversion instructions, A-18
FP status and control reg. (FPSCR), 5-1
FP store instructions, A-21
FPR*n* (floating-point registers 0–31), 1-18
FPSCR instructions, A-18
instructions, 3-13–3-16
arithmetic, 3-14
compare, 3-15
load, 3-21
move, 3-16
multiply-add, 3-14
rounding and conversion, 3-15
status and control, 3-15
store, 3-22
load/store address generation, 3-21
FPR*n* (floating-point registers 0–31), 1-18, 2-3
FPSCR (floating-point status and control reg.), 1-18, 2-3, 5-1
FPSCR (floating-point status and control register)
bit settings, 2-4
FPSCR instructions, A-18
Full-power mode, 9-2
with DPM disabled, 9-3
with DPM enabled, 9-3
Fully static, 9-1

G

G2
overview, 1-15
Global accesses, signaling and snooping, 4-29–4-32
GPR*n* (general-purpose registers 0–31), 1-18, 2-3

H

Hashing functions
primary PTEG, 6-28
secondary PTEG, 6-29
see also Memory management unit (MMU)
HASH*n* (hash address regs., primary/secondary), 2-19, 6-32
HID*n* (hardware implementation registers 0–2), 2-12, 2-15, 2-16
cache control parameters, 4-14–4-17
HID0 register
doze bit, 9-3
DPM enable bit, 9-1, 9-3
HID0 register, nap bit, 9-4
HID1, PLL configuration, 8-4

I

I/O accesses, 8-8
IABR/IABR2 (instruction address breakpoint regs.), 2-11, 2-23–2-24, 10-1
IBAT*n*U/L (instruction block address translation regs. 0–7, upper/lower), 1-5, 2-9, 2-11, 2-20, 2-21
IBCR (instruction address breakpoint control reg.), 2-11, 2-24, 10-2
icbi, 4-20
icbt, 4-20
ICMP (instruction TLB compare register), 2-18, 6-32, 6-34
Illegal instructions, 5-29
IMISS (instruction TLB miss address reg.), 2-18, 6-31, 6-34
Instruction access errors, *see* ISI (instruction storage interrupt)
Instruction address breakpoint interrupt, 5-5, 5-34, 10-1, 10-3, 10-4
Instruction cache, *see* Caches
Instruction latencies, 7-28
see also Execution timing
Instruction set model
overview, 3-10
summary, 3-4
Instruction timing
overview, 1-30–1-31
see also Execution timing
Instruction TLB miss interrupt, 5-5, 5-33
Instructions
branch instructions, A-22
cache control instructions, 4-17–4-21
CSB operations for cache instructions, 4-29
cache management instructions, A-23
classes
defined, 3-5
illegal, 3-5
reserved, 3-5
condition register logical, A-22
e300, instructions not implemented, B-1
floating-point, A-17–A-22
illegal, program interrupt, 5-29
implementation-specific, 3-34
integer, 3-1–3-22, A-15–A-20
load and store, 3-16

byte-reverse instructions, A-20
integer multiple instructions, A-20
string instructions, A-20
memory control, A-23
memory synchronization, A-21
operating environment architecture (OEA), 3-29–3-32
performance monitor, 11-7
PowerPC instructions
complete lists
form (format), A-24
function, A-15
legend, A-35
mnemonic, A-1
opcode, A-8
processor control, A-23
rfci, 5-16
rfi, 5-15
sc, 5-4, 5-31
segment register manipulation, A-24
system linkage, A-22
TLB management instructions, A-24
trap instructions, A-23
user instruction set architecture (UISA), 3-10–3-24
virtual environment architecture (VEA), 3-27
$\overline{int}$ signal, 5-4, 5-12, 5-25
Integer unit (IU), 7-3
execution timing, 7-21
latency, integer instructions, 7-29
Interrupt handling, 5-2
classes of interrupts, 5-2
synchronous exceptions
imprecise, 5-2
precise, 5-2, 5-5
core interface operations, 8-1–8-4, 8-8–8-9
enabling and disabling interrupts and exceptions, 5-14
enabling critical interrupts (MSR[CE] bit), 5-13
enabling external interrupts (MSR[EE] bit), 5-12, 9-5
instruction-related interrupts, 3-9
interrupt modes
FP interrupt mode 0, 5-13, 5-14
FP interrupt mode 1, 5-13
interrupt types, 5-17–5-36
alignment, 5-4, 5-25
critical input interrupt (*cint*), 5-4, 5-13, 5-30
data TLB miss on load, 5-5, 5-33
data TLB miss on store, 5-5, 5-34
decrementer interrupt, 5-4, 5-30
DSI (data storage interrupt), 5-3, 5-23, 10-3, 10-4
external input interrupt (*int*), 5-4, 5-12, 5-25
floating-point unavailable, 5-4, 5-29
instruction address breakpoint, 5-5, 5-34, 10-1, 10-3, 10-4
instruction TLB miss, 5-5, 5-33
ISI (instruction storage interrupt), 5-4, 5-24
machine check, 5-3, 5-13, 5-21, 5-22
performance monitor interrupt, 1-14, 5-33, 11-1, 11-9
program interrupt, 5-4, 5-28
system call, 5-4, 5-31
system management interrupt (*smi*), 5-5, 5-12, 5-36
system reset, 5-3, 5-18
trace interrupt, 5-5, 5-31, 10-3, 10-4
latencies
hard reset and machine check, 5-17
soft reset, 5-17, 5-18, 5-19, 5-20
prefix for interrupt vector offsets, 5-13, 5-15
priorities, 5-5–5-7
process switching guidelines, 5-16
processing of interrupts, 5-8–5-16
front-end actions and state, 5-7
recoverable interrupt indication (MSR[RI]), 5-14, 5-15, 5-16
registers, 5-8–5-14
critical save/restore 0–1 (CSRR0–1), 5-8, 5-10, 5-15, 5-16, 5-17
data address register (DAR), 5-23, 10-2, 10-3
DSI status register (DSISR), 5-1, 10-2, 10-3
floating-point status and control (FPSCR), 5-1, 5-29
FPSCR, 5-1, 5-29
MSR, 5-17
save/restore 0–1 (SRR0–1), 5-8, 5-9, 5-10, 5-14, 5-15, 5-16, 5-17
returning from an interrupt handler, 5-15
returning from critical interrupt, 5-16
Interrupt little-endian mode, 5-12
IQ (instruction queue), 7-9
ISI (instruction storage interrupt), 1-28, 5-4, 5-24

J

JTAG test and debug interface, 1-14, 1-33
JTAG signals, 8-1, 8-3, 8-9

L

Little-endian mode, 5-14
Load/store unit (LSU), 1-1, 7-3
caching-allowed loads/stores, 4-13
execution timing, 7-24
latencies of load and store instructions, 7-32
load/store ordering, 4-14
overview, 1-10
performed loads and stores, definition, 4-13
Logical addresses, translation into physical addresses, 6-1
LR (link register), 2-6
lwarx/stwcx., atomic memory references, 4-14, 8-7

M

Machine check interrupt, 5-3, 5-21, 5-22
  checkstop state, 5-22
  enabling with MSR[ME], 5-13
  SRR1 bit settings, 5-9
MBAR (system memory base address reg.), 2-11, 2-23
Memory accesses, 1-32
Memory management unit (MMU)
  address translation flow, 6-11
  address translation mechanisms, 6-8, 6-11
  block address translation (BAT), 6-8, 6-11, 6-19
  block diagram, 6-5–6-7
  cache locking, 4-34, 4-35, 4-37, 4-40, 4-41, 4-45
  data accesses (DMMU), 6-1
  direct address translation, 6-11, 6-19
  enabling/disabling translation
    data address translation, MSR[DR], 5-13
    instruction address translation, MSR[IR], 5-13
  features summary, 6-2
  hashing functions
    primary/secondary PTEG, 6-28, 6-29
  instruction accesses (IMMU), 6-1
  instructions, 6-16
  interrupts, 6-14
  memory protection, 6-9
  overview, 1-5, 1-11, 1-29
  page address translation, 6-8, 6-11, 6-25
  page history status, 6-10, 6-20–6-23
  page table search operation, 6-25
    software table searches, 6-29, 6-34, 6-35
  physical address generation, 6-1
  registers, 6-16, 6-29–6-33
    data TLB compare register (DCMP), 2-18, 6-32, 6-34
    data TLB miss address reg. (DMISS), 2-18, 6-31, 6-34
    instruction TLB compare register (ICMP), 2-18, 6-32, 6-34
    instruction TLB miss address reg. (IMISS), 2-18, 6-31, 6-34
    primary/secondary hash addr. (HASH1/HASH2), 6-32
    temporary GPRs, enabling with MSR[TGPR], 5-12
  segmented memory model, 6-19
  TLBs (translation lookaside buffers), 5-5, 5-33, 5-34
    description, 6-23
    TLB management instructions, 3-33, A-24
      TLB invalidate (**tlbie** instruction), 6-25, 6-44, A-24
    TLB miss interrupts
      data TLB miss on load, 5-5, 5-33
      data TLB miss on store, 5-5, 5-34
      instruction TLB miss, 5-5, 5-33
Memory model
  access ordering, 4-14
  addressing, 3-7
  data organization, 3-1
  segmented memory, 6-19
    see also Memory management unit (MMU)
  sequential consistency of accesses, 4-13
Memory/cache access attributes (WIMG bits), 4-6–4-8
  caching-inhibited accesses (I bit), 4-7, 6-16
    $\overline{ci}$ internal signal, 8-2
    performance, 7-25
  guarded bit (G bit), 4-7
  memory coherency required (M bit), 4-7
    $\overline{gbl}$ internal signal, 8-2
    timing considerations, 7-25
  performance considerations, 7-24
  W, I, and M bit combinations, 4-8
  write-through mode (W bit), 6-16
    timing considerations, 7-25
    write-through stores, 4-6
    $\overline{wt}$ internal signal, 8-2
Mnemonics, simplified (recommended), 3-33
MSR (machine state register), 1-19, 2-7, 2-11, 5-12
  cache locking, 4-34, 4-36, 4-41
  FP interrupt modes 0–1, 5-14
  MSR[CE] (critical interrupt enable bit), 5-13
  MSR[EE], 9-5
  MSR[POW] (power management enable bit), 9-5
  settings due to interrupts, 5-17

N

Nap mode, 9-2, 9-4

O

OEA, instructions, *see* Instructions, operating environment architecture (OEA)
Operands
  conventions, 3-1
    operand placement and performance, 3-4
Optional instructions, A-35

P

Page address translation
  page address translation flow, 6-25
  page size, 6-19
  see also Memory management unit (MMU)
  selection of page address translation, 6-8, 6-12
  table search operation, 6-25
  TLB organization, 6-24
Page history status, R and C bit recording, 6-10, 6-20–6-23
Page tables
  resources for table search operations, 6-29
  software table search operation, 6-29, 6-34

table search for PTE, 6-25
Parity checking, 8-9
Parity error reporting, 4-14
Performance
characterizing through performance monitor event counting, 1-14, 11-1
performance considerations, memory, 7-24
performance transparent functionality, 9-3
see also Execution timing
Performance monitor, 1-14
Performance monitor APU
event counting, 11-10
chaining counters, 11-11
event types, 11-11–11-13
processor context marking, 11-10
unconditional counting, 11-10
examples of uses, 11-11
instructions, 11-7
interrupt triggered by events, 1-14, 5-33, 11-1, 11-9
registers (PMRs), ??–11-7
PLL (phase-locked loop), 9-2
PLL configuration, HID*n*, 2-15, 8-4
PMC0–3 (performance monitor counter registers), 11-5
PMGC0 (global control register 0), 11-3
PMLCa0–PMLCa3 (performance monitor local control registers A, 0–3), 11-4
Power management
default power state, 9-2
doze mode, 9-2, 9-3
enabling with MSR[POW] bit, 5-12, 9-5
example code, 9-6
full-power mode, 9-2
with DPM disabled, 9-3
with DPM enabled, 9-3
modes, 1-13, 9-3
overview, 9-1
software considerations, 9-5
PowerPC architecture
instruction list, A-1, A-8, A-15
levels of implementation, 1-15
operating environment architecture (OEA), xxiii
overview, 1-15
user instruction set architecture (UISA), xxiii, 2-1
virtual environment architecture (VEA), xxiii
Privilege level
privileged state, *see* Supervisor mode
problem state, *see* User mode
Privilege level (supervisor or user), 5-12
Program interrupt, 1-28, 5-4, 5-28
Program order, definition, 7-2
Protection of memory areas
no-execute protection (N bit), 6-12
options available, 6-9
protection violations, 6-14
PTEGs (PTE groups), 6-25
PTEs (page table entries), 6-25
PVR (processor version register), 2-6

Q

Quiescent state, 9-4
quiesce acknowledge signal (*qack*), 9-4
quiesce request signal (*qreq*), 9-4

R

Real addresses (RA), *see* Memory management unit (MMU)
Reference (R) bit, 6-10, 6-21
checking, 6-36
maintenance recording, 6-10, 6-20–6-22, 6-29
Registers
block address translation registers (BATs)
and cache locking, 4-35, 4-40
branch registers
count register (CTR), 2-6
link register (LR), 2-6
breakpoint registers, 2-11, 2-23–2-27
data address breakpoint (DABR/DABR2), 2-11, 2-25
data address breakpoint control (DBCR), 2-11, 2-27
instruction address breakpoint control (IBCR), 2-11, 2-24
instruction address breakpoint regs. (IABR/IABR2), 2-11, 2-23, 2-24
cache-locking registers
HID*n*, 4-33
configuration registers, 2-6–2-9, 2-12–2-16
hardware implementation regs. (HID*n*), 2-12, 2-15, 2-16
PLL configuration, 2-15
machine state register (MSR), 2-7, 2-11
processor version register (PVR), 2-6
system memory base address (MBAR), 2-11, 2-23
system version register (SVR), 2-11, 2-23
debug
data address breakpoint registers (DABR, DABR2), 10-2
data address control register (DBCR), 10-2
instruction address breakpoint registers (IABR, IABR2), 10-1
instruction address control register (IBCR), 10-2
decrementer register (DEC), 2-10
e300-specific registers, 2-10–2-27
floating-point registers
floating-point regs. 0–31(FPR*n*), 2-3
floating-point status and control reg. (FPSCR), 2-3
general-purpose registers (GPR*n*), 2-3
interrupt handling registers, 5-8–5-14

critical interrupt save/restore regs. (CSRR*n*), 2-10, 2-11, 2-22, 5-8, 5-10, 5-15, 5-16, 5-17
data address register (DAR), 2-10, 5-23, 10-2, 10-3
DSI status register (DSISR), 2-10, 5-1, 10-2, 10-3
floating-point status and control (FPSCR), 5-1
save/restore registers (SRR*n*), 2-10, 5-8, 5-9, 5-10, 5-14, 5-15, 5-16, 5-17
SPRG*n*, 2-10, 2-11, 2-22, 5-11
machine state register (MSR), 4-34
MMU registers, 2-9, 2-18–2-21, 6-29–6-33
data block address translation regs. (DBAT*n*U/L), 2-9, 2-11, 2-20, 2-21
data TLB compare register (DCMP), 2-18, 6-32, 6-34
data TLB miss address reg. (DMISS), 2-18, 6-31, 6-34
hash address regs., primary/secondary (HASH*n*), 2-19
instruction block address translation regs. (IBAT*n*U/L), 2-9, 2-11, 2-20, 2-21
instruction TLB compare register (ICMP), 2-18, 6-32, 6-34
instruction TLB miss address reg. (IMISS), 2-18, 6-31, 6-34
primary/secondary hash addr. (HASH1/HASH2), 6-32
required physical address register (RPA), 2-20
SDR1, 2-9
segment registers (SR*n*), 2-10
performance monitor, ??–11-7
counter registers (PMC0–3), 11-5
global control 0 (PMGC0), 11-3
local control A (PMLCa0–PMLCa3), 11-4
user counter registers (UPMC0–3), 11-7
user global control 0 (UPMGC0), 11-4
user local control A (UPMLCa0–UPMLCa3), 11-5
summary, 2-1
supervisor-level reg. summary, 2-6
time base facility (TBL/TBU)
for reading, 1-19, 2-6
for writing, 2-10
user-level reg. summary, 2-3
XER (32-bit), 2-5
Rename registers
definition, 7-2
operation, 7-16
Reservation station, definition, 7-2
Reservations (memory) with **lwarx** and **stwcx.**, 8-7
Reset
hard reset sequence, 9-1
reset exception, 5-7
settings caused by hard reset, 5-19
soft reset, 5-18, 5-19, 5-20
system reset, 5-3, 5-18
Return from critical interrupt (**rfci**), 5-16
rfci, 5-16
rfi, 5-15
Rotate and shift instructions, 3-12, A-16
RPA (required physical address register), 2-20, 6-33

S

SDR1, 2-9
see also Memory management unit (MMU)
Segment registers (SR*n*), 1-19
SR manipulation instructions, A-24
Segmented memory model, *see* Memory management unit (MMU)
Self-modifying code, 3-17
Sequential consistency of memory accesses, 4-13
Serializing instructions, 7-17
Signals
checkstop, 8-9
cint, 5-4, 5-13, 5-30
clock signals, 8-3
coherency system bus (CSB) internal signals, 8-2
external interrupt signals, 8-3
int, 5-4, 5-12, 5-25
overview, 1-33
*pll_cfg*n, 8-4
qack, 8-3
qreq, 8-3
smi, 5-5, 5-12, 5-36
test interface signals, 8-4
Single-stepping, 10-1, 10-3
trace enable (MSR[SE]), 5-13, 5-32, 10-3, 10-4
Sleep mode, 9-2, 9-4, 9-6
Snooping, 4-29–4-32
core bus interface unit (BIU), 4-2
global signaling and M bit, 7-25
retry, core-initiated, 4-32
see also Caches
coherency
snoop response to CSB transactions, 4-30, 4-31
Soft reset, *see also* Reset, 5-18, 5-19, 5-20
Software debug facilities, *see* Debug facilities
SPR encodings not implemented in e300, B-2
SPRG0–SPRG7, 1-5, 2-10, 2-11, 2-22, 5-11
conventional uses, 5-11
SPRs (special purpose registers), 1-18–1-21
SR*n* (segment registers 0–15), 1-19, 2-10
SRR0–1 (save/restore regs. 0–1), 2-10, 5-8, 5-10, 5-14, 5-15, 5-16, 5-17
bit settings for machine check interrupt, 5-9
bit settings for table search operations, 5-10
stwcx., 8-8
Superscalar, definition, 7-2
Supervisor mode, 5-12
Supervisor-level SPRs, 1-19

SVR (system version register), 2-11, 2-23
Synchronization
  context synchronization, 3-8
  execution of **rfci**, 5-16
  execution of **rfi**, 5-15
  execution synchronization, 3-9
  memory instructions
    UISA, 3-25
    VEA, 3-27
  memory synchronization instructions, A-21
  requirements for setting breakpoints, 10-6
  requirements for special registers and TLBs, 3-32
System call (**sc**), 1-28
  system call interrupt, 5-4, 5-31
System linkage instructions, 3-28, A-22
System management interrupt (*smi*), 1-29, 5-5, 5-12, 5-36, 9-1
System register unit (SRU), 7-3
  execution timing, 7-24
  latency
    CR logical instructions, 7-29
    system register instructions, 7-28

T

Table search operations
  algorithm, 6-25
  software routines, 6-29, 6-34–6-44
  software table search operations
    SRR1 bit settings, 5-10
  table search flow (primary and secondary), 6-27
TBL/TBU (time base facility)
  for reading, 1-19, 2-6
  for writing, 2-10
  time base register, 9-2
  time base/decrementer, 1-14
  time-of-day maintenance, 9-4
TGPR*n* (temporary general purpose regs. 0–3), 6-30
Trace interrupt, 5-5, 5-31, 10-3, 10-4
  tracing facilities, 10-1
Translation lookaside buffers (TLBs), *see* Memory management unit (MMU)
Trap instructions, 3-24
  program interrupt, 5-4, 5-28

U

UISA, instructions, *see* Instructions, user instruction set architecture (UISA)
UPMC0–3 (user performance monitor counter registers), 11-7
UPMGC0 (user global control register 0), 11-4
UPMLCa0–UPMLCa3 (user performance monitor local control registers A, 0–3), 11-5
User mode, 5-1, 5-12
User-level SPRs, 1-18

V

VEA, instructions, *see* Instructions, virtual environment architecture (VEA)
Virtual page number, 6-27

X

XER (32-bit), 2-5